Universität Rostock Traditio et Innovatio

Masterarbeit

Intraoperator-Parallelität frequenter Basisoperationen des wissenschaftlichen Rechnens in SQL

vorgelegt von Dennis Weu
Matr.-Nr. 212206684
am 05.09.2017

am Lehrstuhl für Datenbank- und Informationssysteme
der Universität Rostock

Erstgutachter
Prof. Dr. Andreas Heuer
Universität Rostock
Datenbank- und Informationssysteme

Zweitgutachterin
Prof. Dr. Adelinde Uhrmacher
Universität Rostock
Modellierung und Simulation

Betreuer
Dennis Marten
Universität Rostock
Datenbank- und Informationssysteme

Betreuer
Dr. Holger Meyer
Universität Rostock
Datenbank- und Informationssysteme

Inhaltsverzeichnis

Abstract

Die Erkennung von Intentionen und Aktivitäten im Bereich der smarten Umgebungen erfordert die schnelle und effiziente Auswertung großer Datenmengen. Da die üblicherweise genutzten Statistikprogramme, wie zum Beispiel R, bei Hauptspeichergröße überschreitenden Datenmengen Performanceeinbußen erleiden, ist ein hier gewählter Ansatz die Verarbeitung mit einem parallelen Datenbanksystem. In vielen der verwendeten Algorithmen sind dabei solche Operationen des wissenschaftlichen Rechnens, wie beispielsweise die Matrixmultiplikation, Bestandteil, dessen alleinige Ausführung schon zeitaufwendig ist. Die in vielen parallelen Datenbanksystemen umgesetzte Interanfrage- und Interoperatorparallelität ist in diesen Fällen nicht ausreichend, weshalb hier die weniger implementierte Intraoperatorparallelität untersucht wird. Schwerpunkte sind dabei die Evaluierung einer Partitionierungsstrategie, hinsichtlich der lokal möglichen Berechnungen und Kommunikationskosten, anhand der Matrixmultiplikation. Um die erlangten theoretischen Erkenntnisse und den Stand der Umsetzung in einem aktuellen System miteinander zu vergleichen, wurden die Strategien mit Postgres-XL realisiert und ausgewertet.

Detection of intentions and activities in the context of smart environments requires a fast and efficient evaluation of large amounts of data. Commonly used statistics programs, like R, suffer from performance losses when data sizes become larger than main memory. To address this problem, the use of database systems is proposed. Many used algorithms contain operations of scientific computing, for example matrix multiplication, which are in itself time-consuming. In this case the often available interquery- and interoperator-parallelism is not sufficient. Thus in this work, intraoperator-parallelism is examined, which is used less frequently in current database management systems. We aim at the evaluation of partition strategies, with respect to locally feasible computations and communications costs, based on the matrix multiplication. Furthermore, to compare the achieved theoretical findings with the current state in a nowadays system, the evaluated partition strategies are implemented and analyzed with Postgres-XL.

1 Einleitung

Aktivitäts- und Intentionserkennung mittels Machine-Learning-Algorithmen hat sich, beispielsweise durch die Nutzung von Assistenzsystemen, in vielen Bereichen der Forschung und dem täglichen Leben etabliert.
Dabei sorgen immer leistungsfähigere, kostengünstigere und kompaktere Sensoren dafür, dass große Mengen an Informationen einfach zu sammeln sind. Die Auswertung dieser anfallenden großen Datenmengen übersteigt dabei die Möglichkeiten einzelner Hardwarekomponenten, weshalb eine intensive Untersuchung zur effizienten parallelen Verarbeitung solcher Mengen nötig ist.

Die zum Erkennen von Intentionen und Aktivitäten genutzten Algorithmen basieren, beispielsweise mit der Matrizenmultiplikation oder Vektor-Matrixmultiplikation, auf Konstrukten der linearen Algebra. Dabei werden solche Berechnungen oftmals mit Statistikprogrammen wie R [R C16] durchgeführt, welche jedoch bei Hauptspeichergröße überschreitenden Datenmengen Performanceeinbußen verzeichnen. Um auch diese Größe effizient handhaben zu können, ist die Verwendung von Datenbanksystemen ein hier gewählter Ansatz. Mit der Ausrichtung auf die Verwaltung und Bearbeitung großer Datenmengen, sowie der Möglichkeit auf parallelen Datenbanksystemen die Berechnungen zu verteilen, bieten diese gute Voraussetzungen für die Auswertung von Datenmengen, welche die Hauptspeichergröße überschreiten.

Auf dem Gebiet der parallelen Anfrageverarbeitung mit Datenbanksystemen gibt es zwei Arten, die im Stand der Technik zu finden sind: Interanfrage- und Interoperatorparallelität. Hierbei werden im ersten Fall einzelne Anfragen an verschiedene Knoten geschickt und im zweiten Fall die in einer Anfrage vorhandenen Operatoren verteilt. Allerdings treten in wissenschaftlichen Berechnungen häufiger Operationen auf, wie zum Beispiel die Matrixmultiplikation, welche als Einzelne zu zeitaufwändig sind. Die zwei genannten Arten erreichen in dieser Situation nur eine geringfügige Verbesserung, weshalb die Notwendigkeit besteht, einzelne Operatoren zu parallelisieren. Diese als Intraoperatorparallelität bezeichnete Form ist in parallelen Datenbanksystemen jedoch wenig implementiert.

Entgegen der Interanfrage- und Interoperatorparallelität ist es für die Verteilung des einzelnen Operators nötig, die zu bearbeitende Relation in Partitionen aufzuteilen, auf der dann jeweils eine Operatorinstanz die Berechnung ausführt. In dieser Arbeit sollen daher Partitio-

nierungskonzepte beispielhaft für die Matrizenmultiplikation entwickelt und hinsichtlich der Kommunikationskosten untersucht werden. Anschließend findet ein Vergleich der gewählten Konzepte auf dem parallelen Datenbanksystem Postgres-XL statt.

Diese Masterarbeit gliedert sich in sechs Kapitel. Dabei folgt auf diese Einleitung in Kapitel 2 die Problemstellung, in der genauer die vorliegenden Problematiken und die Zielsetzung dieser Arbeit vorgestellt werden. Mit der kurzen Einführung von SQL und relationalen Datenbanken, sowie bestimmter mathematischer Operatoren und dem Vorwärtsalgorithmus des Hidden Markov Model legt das dritte Kapitel *Grundlegende Konzepte* eine Basis, bevor im Kapitel 4 *Stand der Technik und Forschung* vertiefend bekannte Ansätze und Methodiken Erläuterung finden. Auf Grundlage dieser folgt im Kapitel 5 die *Umsetzung*, in der zunächst im Abschnitt 5.1 ein Matrixrelationenschema ermittelt wird. Dieses bildet die Basis für die Entwicklung von sequentiellen SQL-Anfragen für die vorgestellten mathematischen Operatoren im Abschnitt 5.2. In den Abschnitten 5.4 *Parallelisierung der SQL-Anfragen* und 5.5 *Implementation mit Postgres-XL* werden Partitionierungsstrategien ermittelt, hinsichtlich ihrer Kommunikationskosten bei der Matrixmultiplikation evaluiert und anschließend auf dem System Postgres-XL umgesetzt sowie analysiert. Den Schluss dieser Arbeit bildet das Kapitel 6. Dort werden die Erkenntnisse zusammengefasst, bewertet und auf Basis derer ein Ausblick gegeben.

2 Problemstellung

Die Verarbeitung großer Datenmengen erfordert, unabhängig vom System, eine Parallelisierung, um eine effiziente Auswertung zu ermöglichen. Hierbei gibt es verschiedene Ebenen, auf denen Parallelisierung stattfinden kann, welche im speziellen Kontext der parallelen Datenbanksysteme als Interanfrage-, Interoperator- und Intraoperatorparallelität bezeichnet werden.

Algorithmen des maschinellen Lernens beinhalten häufig Operationen, wie beispielsweise die Matrizenmultiplikation, dessen alleinige Ausführung viel Zeit in Anspruch nimmt. Die bloße Verteilung der im Algorithmus vorhandenen Operationen auf existierende Knoten ist, besonders wenn die weiteren Teilberechnungen verhältnismäßig simpel sind, nicht zielführend, da die Dauer des Gesamtprozesses vom langsamsten Operator abhängt. Diese Form, die Interoperatorparallelität, ist daher nicht ausreichend, um solche Auswertungen effizient zu bearbeiten. Eine mögliche Verbesserung ist hierbei die Intraoperatorparallelität, bei der ein einzelner Operator mit dem Ziel der Antwortzeitverkürzung parallelisiert wird. In den gängigen parallelen Datenbanksystemen ist die Interanfrage- und Interoperatorparallelität umfangreich implementiert, wohingegen die Intraoperatorparallelität nur bedingt Umsetzung findet.
Diese Situation ist unter Anderem dem geschuldet, dass sich für die Parallelisierung eines einzelnen Operators die Problemdimension erhöht. Bei den beiden vorherigen Varianten ist es ausreichend, eine Relation im Gesamten auf Knoten zu verteilen. Für die Intraoperatorparallelität ist es hingegen nötig, eine Relation zu partitionieren und die entstandenen Teilrelationen zu verteilen, damit diese von Operatorinstanzen bearbeitet werden können. Darüber hinaus unterscheidet sich bei bestimmten Operationen, wie zum Beispiel dem Durchschnitt, die Berechnungsvorschrift einer Operatorinstanz von der des Gesamtoperators.

Aufgrund dessen wird im Rahmen dieser Arbeit stellvertretend für eine komplexe Operation die Matrizenmultiplikation und als Algorithmus der Vorwärtsalgorithmus betrachtet. Dabei fordert der große Einfluss von Kommunikationskosten auf die Abarbeitungszeiten in verteilten parallelen Berechnungen die Evaluierung einer Partitionierungsstrategie, die bezüglich der lokalen Berechnungskosten und der nötigen Kommunikation optimiert ist. Das primäre Ziel dieser Arbeit ist es daher, am Beispiel der Matrizenmultiplikation günstige Strategien zu finden. In der Folge bietet eine gute Partitionierung dem parallelen Datenbanksystem die theoretische Basis, effiziente und kostengünstige Abarbeitungspläne zu erstellen. Zur Untersuchung, ob diese Möglichkeiten zur Intraoperatorparallelisierung vom Anfrageplanoptimierer bereits ohne An-

frageanpassung ausgenutzt werden, erfolgt eine Analyse von Laufzeiten und Anfragepläne, welche auf der Umsetzung mit dem parallelen Datenbankclustersystem Postgres-XL basieren.

3 Grundlegende Konzepte

Im Verlauf dieser Arbeit werden Möglichkeiten, Verhalten und Probleme bei der Intraoperatorparallelisierung von häufig im wissenschaftlichen Rechnen verwendeten Basisoperationen mit der Datenbankanfragesprache SQL untersucht. Zum besseren Verständnis findet eine Einführung in die verwendeten Bestandteile von SQL und der zu Grunde liegenden Relationenalgebra statt. Ebenso werden verschiedene, beim wissenschaftlichen Rechnen benutzte, mathematische Operationen erläutert und der Vorwärtsalgorithmus im Rahmen des Hidden Markov Models vorgestellt.

3.1 SQL und relationale Datenbanken

Das Relationenmodell ist eines der am weitesten verbreiteten Datenbankmodelle. Dabei findet es in kommerziellen Datenbanksystemen und der Forschung, aufgrund seiner Einfachheit und Exaktheit, gleichermaßen Verwendung [SSH13]. Hier wird, bezugnehmend auf den Inhalt dieser Arbeit, zunächst kurz der Strukturteil vorgestellt und anschließend mit der Relationenalgebra ein Anfragemodell präsentiert. Darauf folgt die Erläuterung der Umsetzung dargestellter Anfragen. Die Abschnitte 3.1.1 bis 3.1.3 orientieren sich am Buch [SSH13].

3.1.1 Relationenschema

Die Basis im Strukturteil des Relationenmodells bildet das Relationenschema. Ein solches besteht aus einer Menge von Attributen, denen ein Wertebereich zugeordnet ist. Oftmals sind Wertebereiche Standarddatentypen, es können aber jegliche endliche, nicht-leere Mengen sein. Eine bestimmte Instanz eines Schemas wird als Relation bezeichnet. Verglichen mit einer Tabelle sind die Attribute einer Relation die Spaltenköpfe der Tabelle. Der Inhalt sind Werte aus dem entsprechenden Wertebereich des Attributs. Eine Zeile der Tabelle ist ein Tupel der Relation.
Dem Relationenmodell liegt der wichtige Aspekt zu Grunde, dass alles eine Relation ist. Bestimmte Objekttypen, wie beispielsweise Listen oder Arrays, sind nicht vorhanden und können nur über ein entsprechendes und dabei vom Nutzer selbst gewähltes Schema definiert werden. Die gewählte Struktur hat dabei erheblichen Einfluss auf die Formulierung der Anfragen und möglicherweise auch auf die Performance dieser. Hier sollen vor allem wissenschaftliche

Berechnungen untersucht werden, welche Matrizen als Objekttyp verwenden. Daher wird im Abschnitt 5.1 ein dafür geeignetes Schema hergeleitet.

3.1.2 Relationenalgebra

Um mit den Datensätzen zu interagieren ist eine Anfragesprache nötig, etwa die Relationenalgebra, welche verschiedene Operationen zur Interaktion mit Datensätzen bereitstellt. Hier werden die in dieser Arbeit wichtigsten Operationen kurz an einem Beispiel dargestellt: Projektion, Selektion, Verbund, Vereinigung und Umbenennung.

Als Beispiel dienen die zwei nachfolgend dargestellten Relationen `RA` und `RB`. Die angegebenen Attribute sind alle vom Datentyp `integer`.

Relation `RA`

A	**B**
1	2
2	4
3	6

Relation `RB`

B	**C**
2	3
3	6
4	8

Projektion

Die Projektionsoperation kann Attribute einer Relation ausblenden. Als Parameter werden die Attribute angegeben, die im Ergebnis vorhanden sein sollen und die Relation, auf der diese Operation Anwendung findet. Für unsere Beispielrelation `RA` und die Projektion über A ergibt sich folgende Darstellung:

$\pi_A(\texttt{RA})$

A
1
2
3

Selektion

Eine Selektion ermöglicht es, Tupel anhand einer Bedingung auszuwählen. Der Parameter ist hier eine Bedingung, die angibt, wann ein Tupel im Ergebnis erhalten sein soll und die Relation, auf der die Operation durchgeführt wird. Die Selektion mit der Bedingung $A > 2$ auf der Relation `RA` führt zu folgendem Ergebnis:

$$\sigma_{A>2}(\texttt{RA})$$

A	B
3	6

Verbund

Mittels der Verbundoperation werden zwei Relationen miteinander über angegebene Verbundattribute verknüpft. Im Ergebnis bleiben solche Tupel erhalten, bei denen die Werte der Verbundattribute identisch sind. Der Verbund von RA und RB über dem Attribut B erzeugt folgende Ergebnisrelation:

$$\texttt{RA} \bowtie_{RA.B=RB.B} \texttt{RB}$$

A	B	C
1	2	3
2	4	8

Umbenennung

Die Umbenennungsoperation ermöglicht es, Attributnamen zu ändern, um sie zum Beispiel für die Vereinigung vorzubereiten, da diese von den Namen der Attribute abhängt. Im Beispiel werden von der Relation RB die Attribute B und C in A und B umbenannt.

$$\rho_{A \leftarrow B, B \leftarrow C}(\texttt{RB})$$

A	B
2	3
3	6
4	8

Vereinigung

Bei der Vereinigung werden alle Tupel aus den beteiligten Relationen in die Ergebnisrelation übernommen und Doppelte entfernt. Voraussetzung ist hierbei, dass die Anzahl, Namen und Typen der Attribute übereinstimmen. Korrekte Benennung lässt sich jedoch durch die eben vorgestellte Umbenennungsoperation herstellen.

$\texttt{RA} \cup \rho_{A \leftarrow B, B \leftarrow C}(\texttt{RB})$

A	B
1	2
2	4
3	6
2	3
4	8

3.1.3 SQL

SQL ist eine Datenbankanfragesprache, welche weitestgehend die Operationen aus der Relationenalgebra umsetzt. Dabei erfüllt sie bestimmte Kriterien an Anfragesprachen, die in [SSH13] aufgelistet wurden. Unter dem Gesichtspunkt dieser Arbeit ist vor allem die Abgeschlossenheit ein wichtiger Aspekt, da diese sicherstellt, dass jede Operation wieder eine Relation erzeugt. Für die Struktur der Ergebnisrelation ist jedoch allein der Nutzer verantwortlich.

Das Basiskonstrukt, um Abfragen mit SQL zu tätigen, ist der `SELECT-FROM-WHERE`-Block (SFW-Block). Dieser setzt sich, entsprechend des Namens, aus dem `SELECT`-, `FROM`- und `WHERE`-Abschnitt zusammen.

```
1 SELECT projektionsliste FROM relationsliste WHERE bedingungen
```

Programmbeispiel 3.1: SFW-Block von SQL

Im Folgenden werden die SQL-Anfragen zu den in Abschnitt 3.1.2 vorgestellten Operationen aufgeführt.

Projektion, Selektion und Umbenennung

Wie im Programmbeispiel 3.1.3 ersichtlich ist, erfolgt die Projektion im `SELECT`- und die Selektion im `WHERE`-Abschnitt des SFW-Blocks. Ebenfalls im `SELECT`-Abschnitt findet die Umbenennung von Attributen statt.

```
1 SELECT A FROM RA
```

Programmbeispiel 3.2: Projektion mit SQL

```
1 SELECT * FROM RA WHERE A > 2
```

Programmbeispiel 3.3: Selektion mit SQL

```
1 SELECT B AS A, C AS B FROM RB
```

Programmbeispiel 3.4: Umbenennung mit SQL

Verbund und Vereinigung

Die Vereinigung erfolgt zwischen zwei `SFW`-Blöcken über den Operator `UNION`. Beim Verbund findet die Verknüpfung hingegen im `FROM`-Abschnitt der Anfrage statt. Hierbei gibt es mehrere Verbundarten, die mit den Literalen `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `NATURAL JOIN` und `FULL OUTER JOIN` erzeugt werden. Außer beim `NATURAL JOIN`, bei dem dies implizit über gleichbenannte Attribute geschieht, ist es notwendig, mit einer `ON`-Klausel die Verbundattribute anzugeben. Die genannten Verbünde unterscheiden sich dabei hinsichtlich der Tupel, die im Ergebnis enthalten sein sollen. Beim inneren Verbund sind nur Tupel im Ergebnis, bei denen die Werte in den Verbundattributen gleich sind. Beim linken und rechten Verbund sind alle Tupel der linken bzw. rechten Relation im Ergebnis und zusätzlich die zutreffenden Tupel aus der anderen Relation. Der komplette äußere Verbund gibt alle Tupel beider Relationen zurück. Dabei werden Datensätze zu einem Tupel verbunden, wenn die Verknüpfungsbedingung erfüllt ist. Im anderen Fall werden die übrigen Attribute mit NULL-Werten gefüllt.

Im Abschnitt 3.1.2 wurde der innere Verbund vorgestellt, da dieser hier am häufigsten Verwendung finden wird. Programmbeispiel 3.1.3 zeigt die entsprechende SQL-Anfrage.

```
1 SELECT * FROM RA INNER JOIN RB ON RA.B = RB.B
```

Programmbeispiel 3.5: Innerer Verbund mit SQL

Dieser Verbund wird ebenfalls als Equijoin bezeichnet und kann in SQL auch nur mit `JOIN` oder über `[...] FROM RA, RB WHERE RA.B = RB.B` erzeugt werden. Wird im letzten Fall der `WHERE`-Abschnitt weggelassen, so entsteht ein kartesisches Produkt.

Aggregatfunktionen

Zusätzlich zu den präsentierten Operationen ist es möglich, Funktionen anzugeben, die eine Menge von Werten in einen Wert umwandeln. Diese werden als Aggregatfunktionen bezeichnet. In SQL gibt es davon bereits einige Vordefinierte, die im `SELECT`-Teil der Anfrage aufgerufen werden. Bekannte Funktionen sind die Summation (`SUM()`), das Maximum und Minimum (`MAX()` und `MIN()`) und der Durchschnitt (`AVG()`). Neben weiteren vordefinierten Funktionen ist es auch möglich solche über Anfragen selbst zu gestalten.

Aggregatfunktionen können nach [GBL+96] in drei Kategorien eingeteilt werden: distributiv, algebraisch und holistisch. Für die Klassifizierung nehmen wir eine zweidimensionale Menge von Werten $\{X_{i,j}\}$ an.

- **Distributiv:** Eine Aggregatfunktion F ist distributiv, wenn es eine Funktion G gibt, sodass $F(\{X_{i,j}\}) = G(\{F(\{X_{i,j}|i = 1, ..., I; j = 1, ..., J\})\})$. Bekannte Funktionen dieser Kategorie sind `SUM()`, `MIN()`, `MAX()` und `COUNT()`. Außer für `COUNT()` gilt bei den Beispielen sogar $G = F$.

- **Algebraisch:** Die Aggregatfunktion F ist algebraisch, wenn es eine M-Tupel-wertige Funktion G und eine Funktion H gibt, sodass $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j}|i = 1, ..., I, j = 1, ..., J\})\})$. Beispielhaft für diese Kategorie ist die `AVG()`-Funktion.

- **Holistisch:** Eine Aggregatfunktion F ist holistisch, wenn es keine Konstante M gibt, durch die ein M-Tupel die Berechnung charakterisiert. Als Beispiel sei hier der Median genannt.

Zur besseren Einordnung erfolgt eine Beschreibung der Funktionen F, G beziehungsweise H für die Aggregatfunktionen `SUM()`, `COUNT()` und `AVG()`. Hierbei agieren die inneren Funktionen, F und G, auf Partitionen. Im Fall der `SUM`-Aggregation bildet die innere Funktion F Teilsummen für jede Partition. Diese werden im Anschluss von der Funktion G zur Gesamtsumme addiert. Daraus ist ersichtlich, dass F und G hinsichtlich der Berechnungsvorschrift gleich sind und sich nur die Eingabewerte ändern. Bei der `COUNT`-Aggregation sind diese Funktionen nicht gleich. Auf den einzelnen Partitionen zählt die Funktion F die Anzahl der Einträge und G addiert diese anschließend zum Endergebnis. Wichtig für die Kategorie *distributiv* ist demnach, dass die innere Funktion F, welche auf den Partitionen arbeitet, der äußeren Funktion F entspricht. Für die *algebraischen* Aggregatfunktionen gilt dies nicht, wie am Beispiel vom `AVG` gezeigt werden soll. Hier erzeugt die Funktion G für jede Partition ein Tupel aus Summe und Anzahl der vorhandenen Elemente. Die Funktion H addiert und dividiert diese anschließend. Bei dieser Kategorie ist daher die Funktion, welche auf den Partitionen arbeitet, nicht identisch zu der äußeren Funktion F.

Für diese Arbeit von Bedeutung ist die Parallelisierbarkeit von Aggregatfunktionen. Aus der Übersicht wird deutlich, dass distributive Funktionen ohne Veränderung auf Partitionen agieren können. Algebraische Funktionen lassen sich durch ein Umschreiben dieser parallelisieren. Für holistische Funktionen gilt, dass sie nicht in einem Schritt parallelisierbar sind, da dort die Gesamtheit der Daten betrachtet werden muss.

3.2 Mathematische Operationen

In diesem Abschnitt findet eine Vorstellung mathematischer Operationen statt, die im Bereich des maschinellen Lernens Bestandteil vieler Algorithmen sind. Da diese Operationen, auch in Verbindung mit bekannten Verfahren des maschinellen Lernens, in der späteren Umsetzung mittels SQL implementiert und anschließend parallelisiert werden sollen, gilt dieser Abschnitt der mathematischen Präsentation des Skalarproduktes, der Matrixmultiplikation, Transponierung und Faktorisierung. Im Rahmen dieser Arbeit beziehen sich die Operationen und Algorithmen auf Zahlen des reellen Raums $\mathbb{R}$. Die nachfolgenden Einführungen richten sich nach dem Buch [Fis14].

3.2.1 Skalarprodukt

Im reellen n-dimensionalen Raum $\mathbb{R}^n$ wird zwei Vektoren $x = (x_1, ..., x_n)$ und $y = (y_1, ..., y_n)$ die Zahl

$$\langle x, y \rangle = x_1 y_1 + ... + x_n y_n$$

zugeordnet. Die sich ergebende Abbildung

$$\langle\ ,\ \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}, (x, y) \mapsto \langle x, y \rangle$$

heißt kanonisches Skalarprodukt.

3.2.2 Matrixmultiplikation

Das eben vorgestellte Skalarprodukt findet bei der Matrixmultiplikation Verwendung. Dabei definiert sich diese für zwei Matrizen

$$A = (a_{ij}) \in M(m \times n) \text{ und } B = (b_{jk}) \in M(n \times r)$$

als

$$A \cdot B = (c_{ik}) \in M(m \times r) \text{ erklärt durch } c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}.$$

Eine Matrixmultiplikation ist demnach nur möglich, wenn die Spaltenanzahl der Matrix A und Zeilenanzahl der Matrix B gleich sind. Wird eine Zeile der Matrix A als transponierter Spaltenvektor aufgefasst, und eine Spalte der Matrix B ebenfalls als Spaltenvektor, so entspricht dies dem Skalarprodukt.

Blockmatrixdarstellung

Es ist möglich, Matrizen in Blöcke zu zerlegen, die ihrerseits wieder kleinere Matrizen sind. Hierfür ist in Abbildung 3.1 eine beispielhafte Aufteilung einer Matrix in Blöcke dargestellt.

$$\left[\begin{array}{|cc|c|}\hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \\ \hline\end{array}\right]$$

Abbildung 3.1: Aufteilung einer Matrix in Blöcke

Aufgrund der Unterteilungen ist es machbar, auf den entstandenen Teilmatrizen Berechnungen zu formulieren, wie beispielsweise für die Matrixmultiplikation. Dabei muss für die Größe der einzelnen Blöcke, bei $A \cdot B$, die Bedingung gelten, dass die Blockbreite von A der Blockhöhe von B entspricht. Darüber hinaus gilt weiterhin die Bedingung an die Gesamtdimensionen der beiden Matrizen. Sind Beide erfüllt, so gilt für Blockmatrizen dieselbe Berechnungsvorschrift wie für die klassische Matrixmultiplikation, da in dem Fall Elemente als einelementiger Block aufgefasst werden. Der Unterschied ist hingegen das Zwischenergebnis, welches kein skalarer Wert, sondern wiederum eine Matrix ist. Aus diesem Grund ist hier insbesondere auf die Faktorenreihenfolge der einzelnen Teilmultiplikationen zu achten [Str03]. Nachfolgend findet eine schematische Darstellung der Multiplikation zweier Blockmatrizen, mit vier Teilmatrizen, statt.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

3.2.3 Transponierung

Die eben erwähnte Transponierung beschreibt das Tauschen von Zeilen- und Spaltenindizes bei Vektoren und Matrizen. Formal wird dies wie folgt beschrieben:

$$A = (a_{ij}) \in M(m \times n) \Rightarrow A^T = (a'_{ij}) \in M(n \times m) \text{ mit } a'_{ij} := a_{ji}.$$

3.3 Hidden Markov Model

Stellvertretend für Machine-Learning-Verfahren wird hier der frequent genutzte Vorwärtsalgorithmus vorgestellt, welcher auf dem Hidden Markov Model (HMM) basiert. Dieses wird daher zunächst eingeführt. Als Hidden Markov Model wird ein doppelt stochastischer Prozess bezeichnet, bei dem es einen unterliegenden und nicht sichtbaren Prozess gibt, der nur durch weitere stochastische Prozesse, welche eine Sequenz von sichtbaren Symbolen erzeugen, beobachtbar ist [RJ86]. Anwendung findet das HMM zum Beispiel bei der Spracherkennung und Aktivitätserkennung, da hier zu den empfangenen Signalen der Zustand, zum Beispiel gesagter Buchstabe oder Position des Menschen, gesucht wird. Die nachfolgende Einführung richtet sich nach der von L. Rabiner in [RJ86].

Definiert wird das HMM λ durch ein Fünf-Tupel (S, V, A, B, π):

- Eine endliche Menge an diskreten Zuständen $S = \{s_1, s_2, ..., s_n\}$
- Eine endliche Menge an beobachtbaren Symbolen $V = \{v_1, v_2, ..., v_m\}$
- Übergangswahrscheinlichkeiten $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ zwischen den Zuständen
- Wahrscheinlichkeiten $B = (b_{ij}) \in \mathbb{R}^{n \times m}$, um eine Beobachtung in einem bestimmten Zustand zu machen
- Anfangsverteilung $\pi \in \mathbb{R}^n$, die angibt, mit welcher Wahrscheinlichkeit ein Zustand der Anfangszustand ist

Von Bedeutung ist, dass die Übergangswahrscheinlichkeit nur vom aktuellen Zustand abhängt und nicht von den Vorherigen.

Es werden drei Hauptprobleme charakterisiert, deren Lösungen für eine Anwendung bei realen Problemen notwendig sind.

1. **Auswertungsproblem**: Gegeben sei eine Beobachtungssequenz $O = o_1, o_2, ..., o_T$ und ein HMM λ. Wie kann die Wahrscheinlichkeit berechnet werden, dass O von λ produziert wurde?
2. **Aufdecken der versteckten Zustände**: Gegeben sei auch hier eine Beobachtungssequenz $O = o_1, o_2, ..., o_T$ und ein HMM λ. Wie wählen wir eine Zustandssequenz, die optimal ist?
3. **Optimierung der Modelparameter**: Wie kann das HMM angepasst werden, um die Genauigkeit der Wahrscheinlichkeitsberechnung zu maximieren?

Hier wird in Bezug auf die weitere Arbeit nur auf das Auswertungsproblem eingegangen. Ziel ist es, die Wahrscheinlichkeit zu berechnen, dass eine Beobachtungssequenz O mit der Länge T im HMM λ auftritt - ausgedrückt als $\Pr(O|\lambda)$.

Als naiver Weg besteht die Möglichkeit, die Wahrscheinlichkeit für jede mögliche Zustandssequenz der Länge T zu ermitteln. Ausgangspunkt ist die Wahrscheinlichkeit, dass eine Beobachtungssequenz O und eine Zustandssequenz I gleichzeitig auftreten. Ausdrückbar ist dies mit

$$\Pr(O, I|\lambda) = \Pr(O|I, \lambda) \Pr(I|\lambda).$$

Die Wahrscheinlichkeit $\Pr(O|\lambda)$ ist dabei berechenbar durch die Summation von $\Pr(O, I|\lambda)$ über allen möglichen Zustandssequenzen:

$$\Pr(O|\lambda) = \sum_{I \in S^T} \Pr(O|I, \lambda) \Pr(I, \lambda).$$

Zur weiteren Auflösung der Formel ist es nötig, $\Pr(O|I, \lambda)$ und $\Pr(I|\lambda)$ zu ersetzen. Für jede bekannte Zustandssequenz $I = (i_1, \ldots, i_n)$ ist die Wahrscheinlichkeit für die Beobachtungssequenz $O = (o_1, \ldots, o_T)$ darstellbar als

$$\Pr(O|I, \lambda) = b_{i_1}(o_1) b_{i_2}(o_2) ... b_{i_T}(o_T).$$

Die Wahrscheinlichkeit für das Auftreten einer Zustandssequenz I ist

$$\Pr(I|\lambda) = \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} ... a_{i_{T-1} i_T}.$$

Nachteilig ist bei diesem Ansatz die Komplexität von $2T \cdot i$, weil für jeden Zeitpunkt alle Zustände Betrachtung finden und für jeden Summanden $2T$ Berechnungen durchgeführt werden. Es wird deutlich, dass diese Komplexität schon bei kleinen Sequenzen für Größenordnungen sorgt, die eine angemessene Auswertung unmöglich macht. Als alternative Lösung gibt es den Vorwärtsalgorithmus, der im folgenden Abschnitt vorgestellt wird.

3.3.1 Vorwärtsalgorithmus

Der Vorwärtsalgorithmus berechnet $\Pr(O|\lambda)$ mit Hilfe sogenannter Vorwärtsvariablen, die als $\alpha_t(i) = \Pr(o_1, o_2, ..., o_t, i_t = q_i|\lambda)$ definiert werden. Der Algorithmus besteht aus drei Schritten: *Initialisierung*, *Rekursion* und *Terminierung*.

Initialisierung

$$\alpha_1(i) = \pi_i b_i(o_1), 1 \leq i \leq |S|$$

Rekursion

$$\alpha_t(i) = \left(\sum_{j=1}^{|S|} \alpha_{t-1}(j) a_{ji} \right) b_i(o_t), 1 \leq t \leq T,$$

Terminierung

$$\Pr(O|\lambda) = \sum_{j=1}^{|S|} \alpha_T(j)$$

Zum besseren Verständnis wird der Algorithmus anhand eines kurzen Beispiels dargestellt. Wir definieren unser $\lambda = (S, V, A, B, \pi)$ wie folgt:

$$S = \{Sonne, Regen\}$$

$$V = \{nass, trocken\}$$

$$A = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}$$

$$B = \begin{pmatrix} 0.1 & 0.9 \\ 0.95 & 0.05 \end{pmatrix}$$

$$\pi = \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix}$$

Die Beobachtungssequenz O, für die $\Pr(O|\lambda)$ gesucht wird, ist $(nass, nass)$. Die ersten Werte des Vektors α lassen sich über die Initialisierungsformel errechnen.

$$\alpha = \pi \cdot \begin{pmatrix} b_{Sonne}(nass) \\ b_{Regen}(nass) \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix} \cdot \begin{pmatrix} 0.1 \\ 0.95 \end{pmatrix} = \begin{pmatrix} 0.07 \\ 0.285 \end{pmatrix}$$

Im zweiten Schritt werden die Werte von α über die Rekursionsformel ermittelt:

$$\alpha = \begin{pmatrix} (a_1(Sonne)a_{Sonne,Sonne} + a_1(Regen)a_{Regen,Sonne})b_{Sonne}(nass) \\ (a_1(Sonne)a_{Sonne,Regen} + a_1(Regen)a_{Regen,Regen})b_{Regen}(nass) \end{pmatrix} =$$

$$\begin{pmatrix} (0.07 \cdot 0.8 + 0.285 \cdot 0.3) \cdot 0.1 \\ (0.07 \cdot 0.2 + 0.285 \cdot 0.7) \cdot 0.95 \end{pmatrix} = \begin{pmatrix} 0.01415 \\ 0.202825 \end{pmatrix}$$

Abschließend lässt sich die Wahrscheinlichkeit $\Pr(O|\lambda)$ mit der Terminierungsformel berechnen:

$$\Pr(O|\lambda) = 0.01415 + 0.202825 = 0.22.$$

Die gesuchte Wahrscheinlichkeit für die Beobachtungssequenz $(nass, nass)$ im gegebenen HMM λ ist folglich 22%.

4 Stand der Technik und Forschung

Algorithmen des maschinellen Lernens arbeiten oftmals auf einer Menge von Daten, deren Auswertung die Leistungsfähigkeit einer oder weniger Rechner übersteigt. Damit weiterhin eine effiziente Verarbeitung der Daten mittels dieser Algorithmen möglich ist, gibt es den Ansatz, die Ausführung zu parallelisieren. Zur Betrachtung dieser parallelen Auswertung wird hier das parallele Datenbanksystem vorgestellt und von den ähnlichen Systemarten, verteiltes Datenbanksystem und Multidatenbanksystem, abgegrenzt. Vertiefend liegt in dieser Arbeit der Fokus auf den parallelen Datenbanksystemen und der Intraoperatorparallelität, weshalb zugehörige Architekturen, Methoden zur Datenplatzierung, einhergehende Probleme und Standardalgorithmen vorgestellt werden. Abschließend erfolgt die Darstellung des Datenbanksystems PostgreSQL und der parallelen Variante Postgres-XL. Weitere bekannte Ansätze aus der parallelen Verarbeitung von Daten, wie beispielweise MapReduce [DG04] oder die parallele Datenflussprogrammierung, finden hier keine Betrachtung, da im Rahmen des Forschungsprojektes *PArADISE* bereits einige Vorteile bei der Verwendung von SQL identifiziert wurden [MH17]. Neben Vorzügen, welche sich auf die weiteren Forschungspunkte Privatheit und Provenance beziehen, ist der hier Bedeutsame die Möglichkeit mit SQL Algorithmen ohne Anpassung der Anfrage zu parallelisieren. Inwieweit dies auch auf die Intraoperatorparallelität zu trifft, wird im Rahmen dieser Arbeit untersucht. Die nachfolgenden Abschnitte 4.1 bis 4.4 richten sich nach dem Buch [ÖV11].

4.1 Parallele Datenbanksysteme

Im Umfeld der parallelen und verteilten Datenverarbeitung lassen sich drei Systemarten, die aufgrund ihrer verschiedenen Charakterisierung für unterschiedliche Aufgabenbereiche geeignet sind, festmachen: verteilte Datenbanksysteme, Multidatenbanksysteme und parallele Datenbanksysteme.

Eine verteilte Datenbank ist eine Kollektion mehrerer, logisch miteinander verknüpfter Datenbanken, die über ein Netzwerk verteilt sind und über dieses kommunizieren. Die Software, welche diese Datenbanken verwaltbar und gleichzeitig die Verteilung gegenüber dem Nutzer transparent macht, ist das verteilte Datenbanksystem. Bei Multidatenbanksystemen sind die Knoten vollkommen autonom und haben, entgegen den verteilten Datenbanken, kein Kooperationskonzept. Als paralleles Datenbanksystem wird eines auf einem eng gekoppelten Mul-

tiprozessorsystem bezeichnet. Bekannte Architekturen sind hier Shared-Nothing, Shared-Disk und Shared-Memory. Die beiden zuletzt genannten unterscheiden sich insofern von den verteilten Datenbanken, als das die Kommunikation nicht über das Netzwerk geschieht, sondern über die geteilten Ressourcen. Auch wenn Shared-Nothing in dieser Hinsicht einer verteilten Datenbank ähnlich ist, so liegt hier der Unterschied darin, dass die Knoten allesamt homogen sind. Auf die eben genannten Architekturen und deren Mischformen, wie dem Cluster, wird jetzt eingegangen.

4.1.1 Architekturen

Die Architekturen unterscheiden sich hinsichtlich der Art und Weise wie die Hauptkomponenten, Prozessor, Hauptspeicher und Festplatte, über ein schnelles Netzwerk verbunden sind und welche Komponenten sich die Knoten teilen. Abhängig von diesen geteilten Ressourcen gibt es die drei bereits vorher erwähnten Hauptarchitekturen Shared-Nothing, Shared-Memory und Shared-Disk. Eine Mischarchitektur ist das Cluster.

Shared-Memory

Beim Shared-Memory-Ansatz hat jeder Prozessor über eine schnelle Verbindung, zum Beispiel einen Hochgeschwindigkeitsbus, Zugriff auf den Hauptspeicher oder Festplatte. Alle Prozessoren sind dabei unter der Kontrolle von einem Betriebssystem.

Shared-Nothing

In diesem Fall hat jeder Prozessor exklusiven Zugriff auf seinen eigenen Hauptspeicher und eigene Festplatte. Jeder Knoten, bestehend aus Prozessor, Hauptspeicher und Festplatte, wird kontrolliert von einer eigenen Kopie des Betriebssystems. Demnach können diese als lokale Station wie bei einem verteilten Datenbanksystem angesehen werden, weshalb hier viele Lösungen verteilter Datenbanksysteme anwendbar sind.

Shared-Disk

Bei dieser Architektur hat jeder Prozessor über eine Verbindung Zugriff auf alle Festplatten, aber exklusiven Zugang zu seinem Hauptspeicher. Der einzelne Prozessor wird von der eigenen Kopie des Betriebssystems verwaltet. Jedem Prozessor ist es also möglich, Datenseiten von den geteilten Festplatten in seinen eigenen Hauptspeicher zu laden. Dadurch entsteht jedoch die Aufgabe, diesen Cache konsistent zu halten, welche durch einen Lock-Manager übernommen wird.
Vorteile von der Shared-Disk-Architektur gegenüber den Anderen sind die gute Erweiterbarkeit, einfacheres Load Balancing und einfache Migration ausgehend von zentralisierten Datenbanksystemen. Auf der Gegenseite steht die Notwendigkeit von verteilten Datenbankproto-

kollen zur Sperrverwaltung und der mögliche Mehraufwand durch Konsistenzbewahrung der einzelnen Caches.

Cluster

Als Datenbankcluster wird eine Ansammlung von autonomen Datenbanken bezeichnet, wobei die einzelnen Datenbanken von einem handelsüblichen Datenbankmanagementsystem verwaltet werden können. Der primäre Unterschied zu einem parallelen Datenbanksystem auf einem Cluster ist, dass das Datenbankmanagementsystem jedes Knotens als Blackbox fungiert und dadurch parallele Datenmanagementaufgaben über eine Middleware realisiert werden müssen. Cluster können entweder eine Shared-Disk- oder Shared-Nothing-Architektur haben.

4.2 Parallelität von Anfragen

Im Laufe dieses Abschnitts werden verschiedene Herangehensweisen an die parallele Verarbeitung von Anfragen vorgestellt. Hierbei findet eine Unterscheidung in Interanfrageparellelität und Intraanfrageparallelität statt. Der erste Begriff beschreibt die parallele Ausführung von mehreren Anfragen, wodurch der Durchsatz erhöht wird, der Zweite, die Parallelisierung innerhalb einer Anfrage mit dem Ziel der Antwortzeitverkürzung. In einer Anfrage können mehrere Operatoren, z.B. der Selektions- und Projektionsoperator, vorkommen, weshalb eine weitere Unterscheidung zwischen der Inter- und Intraoperatorparallelität vollzogen wird. Im Folgenden erfolgt eine Vorstellung der Interoperatorparallelität und in Bezug auf das Thema dieser Arbeit die Präsentation der Intraoperatorparallelität.

4.2.1 Interoperatorparallelität

Interoperatorparallelität beschreibt die Parallelisierung von verschiedenen Operatoren innerhalb einer Anfrage. Dabei ist es zum einen möglich, dass Operatoren ein Erzeuger-Verbraucher-Verhältnis eingehen, wodurch der verbrauchende Operator bereits Tupel verarbeiten kann, während der Erzeuger noch agiert. Ein visualisierendes Beispiel hierfür ist eine Anfrage, bei der Tupel zweier Relationen selektiert werden, um sie anschließend miteinander zu verbinden (siehe Abbildung 4.1). Hier sind die Selektionen die Erzeuger für den Verbund als Verbraucher. Diese Art ist die Pipelineparallelität. Zum anderen ist es möglich, Operatoren auszuführen, die voneinander unabhängig sind, so wie die Selektionsoperatoren im genannten Beispiel. Genannt wird diese Form Unabhängigkeitsparallelität. Aus dem Beispiel geht ebenfalls hervor, dass die Pipeline- und Unabhängigkeitsparallelität in einer Anfrage anwendbar ist.

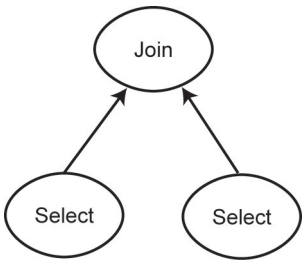

Abbildung 4.1: Operatoren im Erzeuger-Verbraucher-Verhältnis (Grafik aus [ÖV11])

4.2.2 Intraoperatorparallelität

Entgegen dem Verfahren der Interoperatorparallelität wird bei der Intraoperatorparallelität ein einzelner Operator parallelisiert. Hierfür wird dieser in Operatorinstanzen zerteilt, wobei jede Instanz auf einer Partition des gewünschten Datenbestandes arbeitet (siehe Abbildung 4.2).

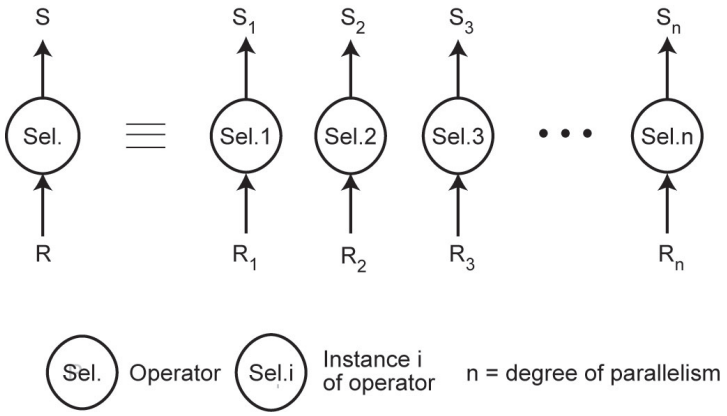

Abbildung 4.2: Zerlegung eines Operators in mehrere Instanzen (Grafik aus [ÖV11])

Im Folgenden werden in Abschnitt 4.3 für die Intraoperatorparallelität nötige Partitionierungsstrategien vorgestellt. Dem folgt die Erläuterung von Problemen in 4.4, die insbesondere im Kontext dieser Parallelisierungsart auftreten, bevor in 4.5 bekannte Standardalgorithmen Beschreibung finden.

4.3 Datenplatzierung

Im Vergleich zu einem zentralisierten Datenbanksystem ergeben sich bei parallelen Datenbanken durch die Verwendung mehrerer Knoten offensichtliche Unterschiede bei der Datenhaltung. Grundlegend ergibt sich die Frage, wo eine Relation gespeichert wird und ob dies als Gesamtheit erfolgen soll. Jenes kann an einem oder mehreren Orten stattfinden.

Alle Daten an einem Ort zu speichern, würde am Ende einem zentralisierten Datenbanksystem entsprechen und die restliche Menge der Knoten unbenutzt lassen. Dementsprechend ist es sinnvoll, die vorhandenen Daten zu verteilen. Wenn jedoch alle Tupel einer Relation auf einem Knoten gespeichert werden, kommt es zu einem Flaschenhals, wenn sich die Mehrheit aller Anfragen an diese Tabelle richtet und dafür sorgt, dass viele Knoten nicht ausgelastet sind. Zur Lösung dieses Problems werden die Tabellen zerteilt und diese Abschnitte auf verschiedene Knoten verteilt. Näher wird darauf im nachfolgenden Abschnitt 4.3.1 Partitionierung eingegangen.

4.3.1 Partitionierung

Wie einleitend beschrieben, ist Partitionierung die Zerteilung einer Relation in Partitionen mit der anschließenden Verteilung dieser auf mehrere Knoten. Ziel ist hierbei, durch die Auslastung vieler Knoten, die Parallelität zu erhöhen und damit die Antwortzeit zu verkürzen. Im Umfeld der verteilten Datenbanksysteme gibt es für das ähnliche Prinzip den Begriff der Fragmentierung, welcher in horizontale und vertikale Fragmentierung unterteilt wird. Dort entscheidet der Datenbankadministrator selbst über Kriterien wie Anfragehäufigkeit einer Relation und eines Attributs über die Fragmentierung und Verteilung. Hier wird eine alternative Lösung vorgestellt: die vollständige Partitionierung. Bei dieser wird jede Relation horizontal, also tupelweise, über alle Knoten des Systems verteilt. Die drei Basisstrategien sind dabei Round-Robin sowie Bereichs- und Hashpartitionierung.

Round-Robin

Ausgehend von n Partitionen wird das i-te Tupel einer Relation auf die Partition ($i \mod n$) gelegt. Diese Strategie ist eine der einfachsten und erzeugt sogleich eine uniforme Verteilung der Daten.

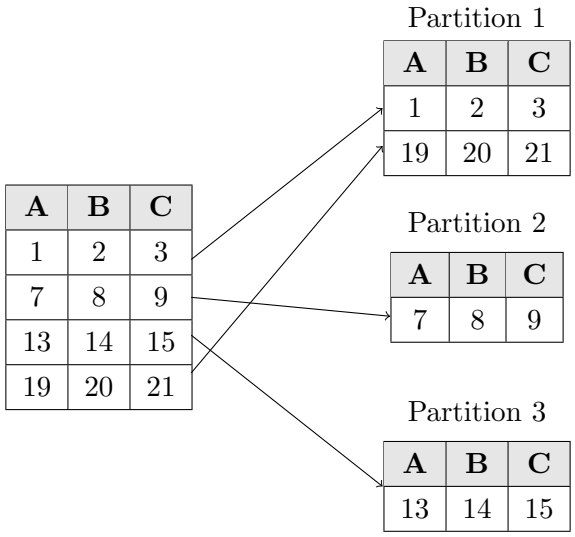

Abbildung 4.3: Round-Robin

Bereichspartitionierung

Bei diesem Verfahren werden Tupel anhand des Wertebereichs eines oder mehrerer Attribute einer Partition zugewiesen. Tupel mit Werten des gleichen Bereichs werden hierbei derselben Partition zugeordnet.

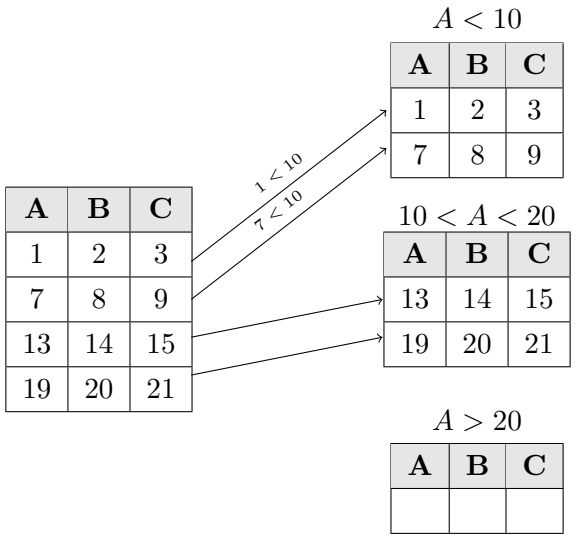

Abbildung 4.4: Bereichspartitionierung

Im Abschnitt 3.2.2 wurden bereits Blockmatrizen vorgestellt. Hierbei handelt es sich im Grunde um eine Partitionierungsform der Mathematik, die mit der hier vorgestellten Bereichspartitionierung abbildbar ist, da ein Block einem Spalten- und Zeilenbereich einer Matrix entspricht. Darüber hinaus findet diese Partitionierungsform im Kontext der parallelen Matrixmultiplikation in der Veröffentlichung [GW97] Benutzung, um die Fälle der *Basic Linear Algebra Subprograms* (BLAS) zu realisieren. Wie diese Partitionierungsstrategie in SQL umsetzbar ist, wird im Abschnitt 5.4.1 beschrieben.

Hashpartitionierung

Für diese Form der Partitionierung wird einem Attribut eine Hashfunktion zugewiesen, welche die Partitionsnummer ausgibt. Folgendes Beispiel zeigt die Verteilung mit der Hashfunktion $h(a) = a \mod 3$, $a \in Dom(A)$.

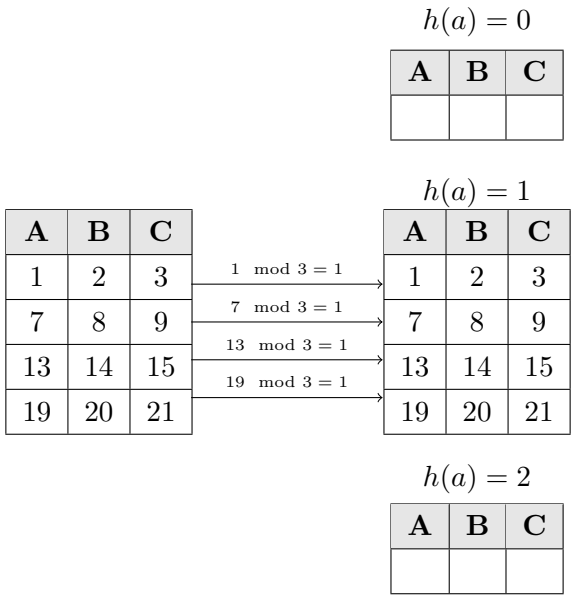

Abbildung 4.5: Hashpartitionierung

Die Beispiele verdeutlichen den starken Einfluss des gewählten Bereiches und gewählter Hashfunktion auf die Größe der Partition. Die Partitionierung hat dabei direkte Konsequenzen für die parallele Ausführung, sodass beispielsweise bei der in 4.5 gezeigten Hashpartitionierung ein Knoten alle Last tragen würde. Es ist folglich für eine gewünschte Aufgabe nötig, die beste Strategie und zugehörige Funktion zu elaborieren.

Überlappende Partitionierung

Bei den bisherigen Partitionierungsarten wurde nicht betrachtet, wie die Berechnung auch beim Ausfall eines Knotens weiterhin sichergestellt werden kann, was offensichtlich nur über replizierte Daten, also dem Anlegen von Kopien, passieren kann. Die überlappende und verkettete Partitionierung sind zwei Möglichkeiten, diese Sicherheitskopien so zu verteilen, dass auch bei Knotenausfällen noch eine Berechnung in angemessener Zeit möglich ist.

Bei der überlappenden Partitionierung werden die Sicherheitskopien in (Anzahl Knoten - 1) Teile zerlegt. Diese werden dann jeweils auf die übrigen Knoten, also die, auf denen nicht die primäre Kopie liegt, verteilt (siehe Abbildung 4.6). Im Fehlerfall wird die Last auf die Knoten mit der Sicherheitskopie geschoben. Beim Ausfall von mehr als einem Knoten ist jedoch keine Weiterführung der Berechnung möglich.

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy		$r_{1.1}$	$r_{1.2}$	$r_{1.3}$
	$r_{2.3}$		$r_{2.1}$	$r_{2.2}$
	$r_{3.2}$	$r_{3.3}$		$r_{3.1}$

Abbildung 4.6: Überlappende Partitionierung (Grafik aus [ÖV11])

Verkettete Partitionierung

Eine andere Lösung ist die verkettete Partitionierung. Anstatt die Sicherheitskopien weiter zu zerteilen, werden diese im Gesamten auf die anderen Knoten so verteilt, dass jeweils der Nachbarknoten diese erhält (siehe Abbildung 4.7). Die Hauptidee dahinter ist, dass die Wahrscheinlichkeit für den Ausfall zweier benachbarter Knoten wesentlich geringer ist, als die für den Ausfall zweier beliebiger Knoten.

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	r_4	r_1	r_2	r_3

Abbildung 4.7: Verkettete Partitionierung (Grafik aus [ÖV11])

4.3.2 Replikation

Datenreplikation bezeichnet die Lagerung von gleichen Datenelementen auf mehreren oder allen Knoten. Das Ziel ist eine verbesserte Verfügbarkeit, höhere Ausfallsicherheit und Verringerung der Kommunikationskosten, die beim Verschicken von Daten auftreten. Im Zuge dessen ist es nötig, Protokolle einzuführen, die dafür sorgen, dass die Datenelemente in den Replika konsistent sind. Solche Protokolle unterscheiden sich zum einen in der Art und Weise, wann sie die Kopien aktualisieren: Entweder noch vor dem Commit einer einzelnen Transaktion, bezeichnet als Eager (dt. eifrig) oder nach dem Commit, was als Lazy (dt. faul) bezeichnet wird. Zum anderen findet eine Unterteilung bezüglich des Ortes statt, an dem eine Update-Operation ausgelöst werden kann - zentral an einem Ort oder verteilt an Mehreren.

4.4 Auftretende Probleme

In diesem Abschnitt werden Probleme, die durch ungünstige Partitionierungen und Szenarien auftreten können, diskutiert und mit Bezug auf diese Arbeit eingeordnet.
Das erste Problem ist mit der Lastbalancierung die Fragestellung, wie ankommende Anfragen auf die vorhandenen Ressourcen verteilt werden, um den möglichst größten Zeitgewinn zu erlangen. Für gewöhnlich ist dies dann, wenn alle Knoten gleich stark ausgelastet sind. In diesem Zusammenhang entsteht mit der Datenschiefheit (engl. data skew) ein weiteres Problem.

4.4.1 Lastbalancierung

Da die Antwortzeit eines parallelen Systems durch den langsamsten der laufenden Prozesse bestimmt wird, hat die Lastbalancierung einen enormen Einfluss auf die Performance eines solchen Systems.
Bei der eingeführten Interoperatorparallelität bearbeitet ein Knoten einen Operator, wobei Operatoren eine unterschiedliche Komplexität aufweisen können. Wenn ein Operator deutlich komplexer ist als die anderen, muss seine Antwortzeit reduziert werden. Dies kann beispielsweise über Intraoperatorparallelität geschehen. Gleichzeitig ist es weiterhin nötig, eine gleichmäßige Verteilung der Last auf alle Knoten zu realisieren, um den Durchsatz zu maximieren. Diese eben beschriebene Schiefheit in der Belastung einzelner Knoten entsteht nicht nur durch die unterschiedliche Komplexität der Operatoren, sondern auch durch ungünstige Verteilung der Daten. Dies wirkt sich vor allem bei der Intraoperatorparallelität aus, weshalb dies hier genauer betrachtet wird.

4.4.2 Datenschiefheit

Bei der Intraoperatorparallelität bearbeiten Operatorinstanzen die ihnen zugeordnete Partition. Obwohl die Komplexität dieser Instanzen auf jeder der Teilmengen gleich ist, kann es

hier zu Schiefheit kommen, wenn die Partitionen unterschiedlich groß sind. Dabei wird die Datenschiefheit weiter kategorisiert in Tupelplatzierungs-, Selektivitäts-, Umverteilungs- und Verbundschiefheit [HM00]. Jede dieser Arten tritt zu unterschiedlichen Zeitpunkten in oder vor einer Berechnung auf.

Tupelplatzierungsschiefheit beschreibt den Sachverhalt, wenn ein Knoten viele Tupel und ein anderer wenige Zeilen zugeordnet bekommt. Bedingt wird dies durch nicht uniforme Daten und die gewählte Partitionierungsstrategie. Als Beispiel kann hier eine Relation angesehen werden, in der Studenten mit ihrem Studienfach gespeichert sind. Bei einer Bereichs- oder Hashpartitionierung über das Studienfachattribut werden manchem Knoten viele Tupel zugeordnet, wie beispielsweise dem, der die Wirtschaftwissenschaftsstudenten verwaltet. Andere Knoten sind nur für wenige Tupel zuständig, beispielsweise der für die Informatikstudenten. Eine einfache Aggregatfunktion wie `COUNT` dauert folglich auf dem Wirtschaftwissenschaftsstudentenknoten länger, als auf dem der Informatikstudenten. Diese Tupelplatzierungsschiefheit ist auch in den Abbildungen aus 4.3.1 erkennbar.

Selektivitätsschiefheit tritt bei der Verarbeitung auf, wenn eine Selektion auf einem der Knoten viele und auf einem Anderen wenige Tupel auswählt.

Umverteilungsschiefheit entsteht beim erneuten Verteilen zwischen zwei Operatoren. Da hier die gewählte Verteilungsart den Schiefheitsgrad bestimmt, ist dies vergleichbar mit der bereits beschriebenen Tupelplatzierungsschiefheit.

Verbundschiefheit entsteht, wenn die Verbundselektivität auf den einzelnen Knoten unterschiedlich ist. Folglich kann dies durch Tupelplatzierungsschiefheit hervorgerufen werden.

Die Auswirkungen von Datenschiefheit können bei der Intraoperatorparallelität im schlimmsten Fall, wenn ein Knoten alle Tupel bearbeiten muss, dazu führen, dass keine Parallelisierung stattfindet. Um die initiale Tupelplatzierungsschiefheit zu verhindern, werden im Abschnitt 5.4.1 des Umsetzungskapitels einzelne Partitionierungsstrategien unter anderem hinsichtlich dieser Problematik betrachtet.

4.5 Intraoperatorparallelität von Aggregatfunktionen

Aggregatfunktionen sind nicht nur in allen Datenbankbereichen ein häufiger Bestandteil von SQL-Anfragen, sondern auch in Algorithmen und Berechnungen des maschinellen Lernens. Daher soll hier einführend auf die in [SN95] vorgestellten grundlegenden Techniken und Ansätze eingegangen werden, welche eine Intraoperatorparallelisierung von Aggregatfunktionen ermöglichen. In der Veröffentlichung werden zwei bereits bekannte Algorithmen analysiert: der zen-

trale Zwei-Phasen-Algorithmus und der Repartitionierungsalgorithmus. Auf Basis der festgestellten Schwächen fand eine Vorstellung des adaptiven Zwei-Phasen- und ebenfalls adaptiven Repartitionierungsalgorithmus statt. Ziel dieses Abschnittes ist die kurze Präsentation der Ansätze und auftretenden Probleme, jedoch nicht die vollständige Darstellung der einzelnen Algorithmen. Der Fokus liegt hier nur auf den im SQL:92-Standard (siehe [Sql]) vorhandenen, distributiven Aggregatfunktionen (siehe Abschnitt 3.1.3) `COUNT`, `MAX`, `MIN`, `SUM` und der algebraischen Funktion `AVG`.

Zwei-Phasen-Algorithmus

Ein intuitiver Ansatz bei der Berechnung eines Aggregats ist der zentrale Zwei-Phasen-Algorithmus, wobei die vorliegenden Daten nach dem Round-Robin-Prinzip verteilt sind. Hier berechnet in der ersten Phase jeder Knoten den lokalen Aggregatwert, beziehungsweise im Fall des Durchschnitts die Summe und Anzahl der Tupel einer Partition. Dieses Vorgehen führt nur dann zu einem einzelnen Zwischenergebnistupel pro Knoten, wenn das Gesuchte ein skalares Aggregat ist - also global nur ein einziges Ergebnistupel entsteht. Über die `GROUP BY`-Klausel einer Aggregatanfrage können jedoch Gruppen gebildet werden, über die aggregiert wird. Eine solche Anfrage ist zum Beispiel das Bilden des Durchschnittsalters nach Studiengang. Hierbei entspricht das Studienfach dem Gruppierungsattribut. Es wird deutlich, dass hier der Koordinator zum Flaschenhals werden kann, wenn es viele verschiedene Studienfächer gibt und somit viele Zwischenergebnistupel entstehen. Um dieses Problem zu beheben, kann die zweite Phase ebenfalls parallelisiert werden, indem die entstandenen lokalen Aggregattupel nach den Gruppenwerten hashpartitioniert und alle Knoten gemeinsam den globalen Aggregatwert bilden. Bekannt ist dieses Vorgehen als Zwei-Phasen-Algorithmus. Durch die Round-Robin-Verteilung ergibt sich jedoch weiterhin, dass im schlimmsten Fall eine Gruppe auf jedem Knoten aggregiert wird. Dadurch sind die Zwischenergebnisse größer und die zweite Phase länger.

Repartitionierungsalgorithmus

Ein anderer standardmäßiger Ansatz, welcher das zuletzt beschriebene Problem löst, ist der Repartitionierungsalgorithmus. Bei diesem findet vor der lokalen Aggregationsphase eine Hashpartitionierung auf die vorhanden Knoten anhand der Gruppenwerte statt, wodurch sich alle Tupel einer Gruppe nur auf einem Knoten befinden können. Es wird deutlich, dass dieser Algorithmus bei wenig verschiedenen Gruppenwerten nicht alle Knoten auslastet. Bei beispielsweise einem einzigen Studienfach wäre nur ein Knoten belastet und die Berechnung würde zu einer sequentiellen entarten.

Adaptiv

Beide Algorithmen weisen Probleme bei verschiedenen Szenarien auf. Diese entstehen beim Zwei-Phasen-Algorithmus, wenn es viele Gruppen gibt und beim Repartitionierungsalgorithmus, wenn die Anzahl gering ist. Daher werden in [SN95] adaptive Varianten der beiden Al-

gorithmen vorgeschlagen, die je nach Belastung von dem Einen in den Anderen wechseln.

Die obige Ausführung zeigt den fundamentalen Wert günstiger Partitionierungsstrategien. Weiterhin wird die Zweistufigkeit von Aggregatfunktionen erneut, wie auch in den Formeln von 3.1.3, deutlich. Die erste Stufe ist dabei die lokale Aggregation auf Partitionen, gefolgt von der zweiten Stufe mit der globalen Aggregation.

4.6 PostgreSQL

PostgreSQL ist ein freies objektrelationales Datenbankmanagementsystem, das auf dem von der University of California geschriebenen Paket, POSTGRES, basiert. Genutzt ist das System im kommerziellen und universitären Umfeld gleichermaßen. Gründe sind hierfür unter anderem, dass PostgreSQL als Open-Source-Projekt frei nutzbar ist, viele Werkzeuge unterstützt werden und eine gute Anbindung an viele Programmiersprachen bietet. Darüber hinaus wird der SQL-Standard zu großen Teilen unterstützt und es existiert eine umfangreiche Dokumentation. In dem hiesigen Kontext ist von Bedeutung, dass es mit Postgres-XL eine auf PostgreSQL basierende Lösung für Clustersysteme gibt. Zunächst soll aber mit dem Abschnitt 4.6.1 auf die Verarbeitung paralleler Anfragen mittels PostgreSQL Version 9.6 eingegangen werden [Pos].

4.6.1 Parallele Anfrage

Seit der Version 9.6 bietet PostgreSQL die Möglichkeit, bestimmte Anfragen auf mehreren Prozessoren gleichzeitig auszuführen. Hierbei prüft der Anfrageplanoptimierer, ob die Verwendung eines parallelen Anfrageplans einen Vorteil gegenüber dem standardmäßig sequentiellen Plan verspricht.

Funktionsweise

Wenn sich der Optimierer für einen parallelen Plan entscheidet, so wird ein Plan generiert, in dem es einen Sammelknoten (engl. Gathernode) gibt. Alle Punkte, die nach diesem Knoten im Plan stehen, werden parallel verarbeitet. Einem Plan mit hohem Parallelitätsgrad steht ein Sammelknoten voran.
Beim Erreichen eines Sammelknotens werden mehrere Prozesse gestartet, wobei die maximale Anzahl vom Administrator festlegbar ist. Jeder von den erfolgreich gestarteten Prozessen arbeitet dann den gleichen Teilplan, der nach dem Sammelknoten steht, ab. Dieser muss so modifiziert sein, dass jeder Arbeitsprozess eine Partition der Daten bearbeitet. Ohne diese Bedingung würde die Parallelisierung zu keiner Beschleunigung führen. Somit generiert jeder Prozess aus einer Teilmenge der Input-Daten wiederum eine von den anderen Teilmengen verschiedene Teilmenge der Output-Daten. Diese modifizierten, parallelen Pläne gibt es für Scans, Verbünde und Aggregationen, wovon letzterer hier vorgestellt wird.

Die parallele Aggregation besteht aus zwei Phasen. In der Ersten erstellt jeder der aktivierten Arbeitsprozesse auf seiner zugeordneten Partition das gewünschte Aggregat und liefert das Ergebnis an den Leadworker, welcher einer der aktivierten Worker ist. Dieser setzt die Teilergebnisse zum Endaggregat zusammen. Dies ähnelt dem in Abschnitt 4.5 beschriebenen Zwei-Phasen-Algorithmus und weist die gleiche Schwäche auf. Sind die Teilergebnisse groß, ist der Führungsprozess stark belastet. PostgreSQL erkennt dies und benutzt in diesem Fall einen sequentiellen Plan für die gesamte Anfrage.

Fazit

PostgreSQL bietet mit der umfangreichen Unterstützung des SQL-Standards, guten Dokumentationen und seiner aktiven Gemeinschaft, gute Voraussetzungen für die Nutzung im universitären und wirtschaftlichen Umfeld. Mit der neuen Erweiterung ist es jetzt möglich, alle Prozessorkerne auszunutzen und somit Parallelität in die sonst sequentielle PostgreSQL-Umgebung zu bringen. Darüber hinaus bildet PostgreSQL die Grundlage für das im nächsten Abschnitt vorgestellte System Postgres-XL.

4.7 Postgres-XL

Postgres-XL ist eine verteilte bzw. parallele Ausprägung von PostgreSQL. Dabei steht das *XL* für *eXtensible Lattice* (dt. erweiterbares Gitter) und beschreibt damit eine Kerneigenschaft von Postgres-XL: Skalierbarkeit. Die Nähe zu PostgreSQL ermöglicht die Einbindung von bereits bestehenden PostgreSQL-Datenbanken.

Mit Postgres-XL ist es möglich, ein Datenbankcluster zu erstellen, bei dem die Datenbanken und Relationen partitioniert und repliziert auf verschiedenen Knoten liegen. Dabei erfüllt es die ACID-Eigenschaften über das *Multi-Version Concurrency Control* Protokoll. Es ist also gewährleistet, dass im gesamten Cluster konsistente Daten vorhanden sind.

Im Nachfolgenden werden die einzelnen Komponenten eines Postgres-XL Systems mit ihren jeweiligen Funktionen vorgestellt. Die zwei Abschnitte 4.7.1 und 4.7.2 richten sich nach [Theb].

4.7.1 Komponenten

Das System besteht aus drei Hauptkomponenten: dem globalen Transaktionsmanager, dem Koordinator und den Datenknoten. Die grundlegende Architektur eines Postgres-XL Cluster ist in Abbildung 4.8 zu sehen.

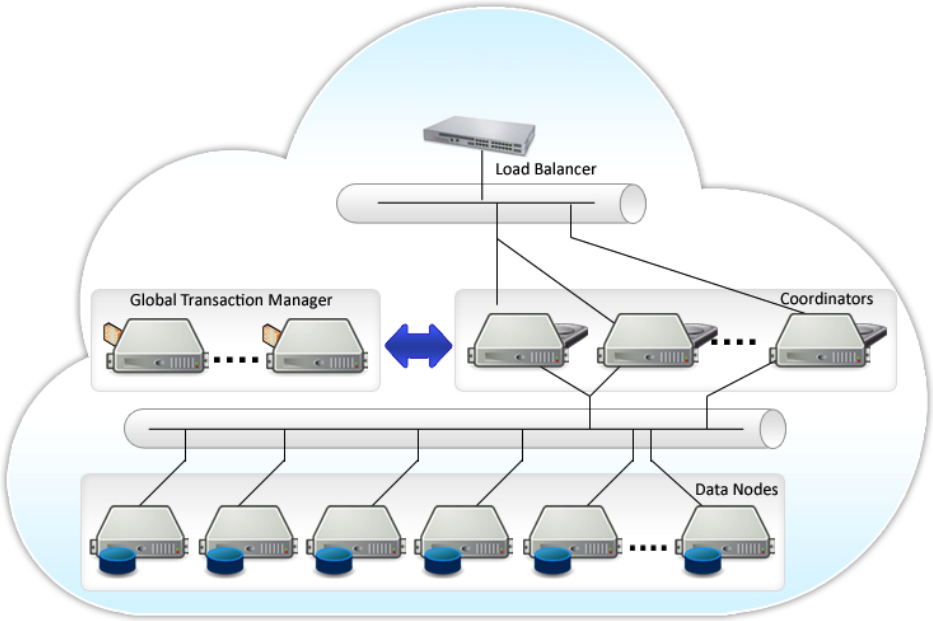

Abbildung 4.8: Postgres-XL Architektur (Bild aus [PX])

Globaler Transaktionsmanager

Der globale Transaktionsmanager (GTM) sorgt für konsistente Transaktionsverarbeitung und kontrolliert die Sichtbarkeit von Tupeln. Um die Konsistenz zu gewährleisten, benutzt der GTM das oben erwähnte *Multiversion Concurrency Control* Protokoll, welches in [BG83] näher beschrieben wird.

Koordinator

Der Koordinator dient als Schnittstelle zwischen den Anwendungen und der Datenbank. Dabei verhält sich der Koordinator wie ein normaler PostgreSQL-Prozess, jedoch speichert er keine Daten.

Datenknoten

Die Datenknoten speichern Relationen. Dabei ist es möglich, diese verteilt, also partitioniert, auf mehreren Knoten zu lagern oder auf alle Knoten zu replizieren. Der jeweilige Datenknoten kennt nur seine lokalen Daten.

4.7.2 Ziele

Das primäre Ziel von Postgres-XL ist Datenbankskalierbarkeit mit gleichzeitiger Erfüllung der ACID-Eigenschaften bei allen Arten von Datenbankbelastungen. Um dies zu realisieren, soll das System mehrere Eigenschaften erfüllen:

1. Transaktionen können von mehr als einem Server angenommen werden.
2. Jeder Koordinator liefert den anfragenden Applikationen einen konsistenten und transparenten Blick auf die Datenbank.
3. Datenknoten können miteinander kommunizieren, um Anfragen effizient und parallel abzuarbeiten.
4. Tabellen können verteilt und repliziert werden, wobei die Verteilung gegenüber der Applikation transparent sein soll.
5. PostgreSQL-Schnittstellen sind kompatibel.
6. Die Struktur von SQL-Anfragen hängt nicht von der tatsächlichen, möglicherweise verteilten und replizierten, Speicherung ab.

Im Rahmen dieser Arbeit sind vor allem die Eigenschaften 3 und 5 bedeutungsvoll, da sie durch die Verteilung und Interknoten-Kommunikation den Grundstein für eine mögliche Intraoperatorparallelisierung von SQL-Anfragen legen.

4.7.3 Erweiterungen gegenüber PostgreSQL

Postgres-XL verbessert, laut den Entwicklern, mehrere Punkte gegenüber dem nicht parallelen Datenbanksystem PostgreSQL. Dazu gehört, neben der aus dem Namen ableitbaren Erweiterbarkeit und Skalierbarkeit, unter anderem auch die massive parallele Verarbeitung (kurz MPP für engl. Massively Parallel Processing). Zu diesem Punkt gehört die Multiknotenverarbeitung, Drei-Phasen-Aggregation, automatische Datenumordnung und kooperative Scans, welche im Folgenden kurz erläutert werden [2nd].

Die Multiknotenverarbeitung ermöglicht das Ausführen von SQL-Anfragen auf mehreren Knoten gleichzeitig, wodurch die Arbeitslast gleichmäßig im Cluster verteilt wird. Durch die drei Phasen Aggregation soll der Kommunikationsaufwand verringert werden, indem die Aggregation parallel und möglichst weit unten realisiert wird. Bei komplexen SQL-Anfragen findet eine automatische Umordnung der Daten statt, um diese zu ermöglichen. Die kooperativen Scans verhindern doppeltes Lesen der gleichen Daten [2nd].

4.7.4 Ausnutzung der parallelen Anfrage von PostgreSQL

Im Abschnitt 4.6.1 wurde die parallele Anfrageverarbeitung von PostgreSQL vorgestellt, die es ermöglicht, ankommende Anfragen auf mehreren Prozessorkernen zu verarbeiten. Diese Entwicklung kann auch Postgres-XL seit der Version 9.15 ausnutzen, da die einzelnen Knoten normale PostgreSQL-Instanzen sind. Das bedeutet, dass zunächst die Arbeit an vorhandene

Datenknoten verteilt wird. Jeder dieser Knoten verteilt die ankommende Arbeit anschließend an die vorhandenen Prozessorkerne (siehe Abbildung 4.9) [Deo].

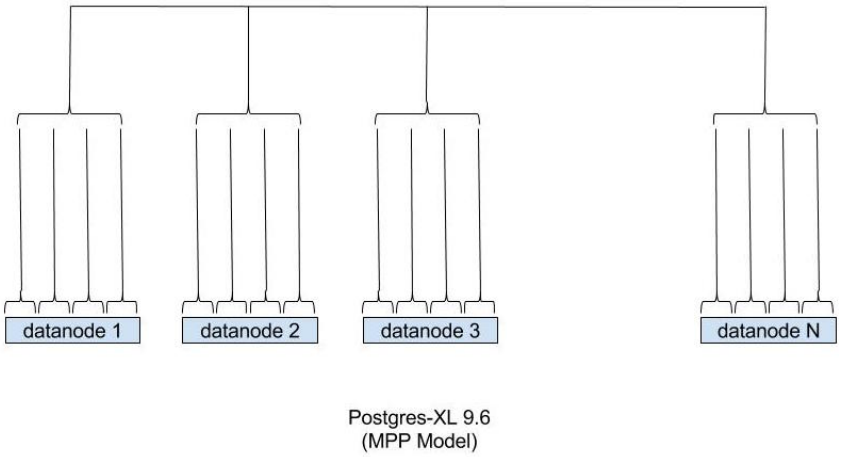

Abbildung 4.9: Postgres-XL mit Paralleler Anfragetechnik von PostgreSQL (Bild aus [Deo])

Fazit

Postgres-XL öffnet mit der Partitionierung und Verteilung von Relationen auf verschiedene Knoten und der Knoten-zu-Knoten-Kommunikation die Tür zur Intraoperatorparallelität, bei der Operatoren in Instanzen zerlegt werden, um auf den einzelnen Partitionen zu agieren (siehe Abschnitt 4.2.2). Als Nachfahre von PostgreSQL bietet es den gleichen Unterstützungsumfang des SQL-Standards, profitiert von deren Weiterentwicklungen und ermöglicht die Nutzung von bekannten Anbindungen wie dem Java-Database-Connectivity-Treiber (JDBC-Treiber). Auf Grund dieser Bedingungen ist Postgres-XL ein guter Kandidat für Intraoperatorparallelität mit einem Datenbankmanagementsystem.

4.8 Fazit

In den vorherigen Abschnitten wurden grundlegende Architekturen, Techniken, Probleme und Standardvorgehensweisen einführend erläutert, um eine Basis für das kommende Umsetzungskapitel zu legen. Hier soll daher zusammenfassend dargestellt werden, welche Punkte aus diesem Kapitel insbesondere im Folgenden Verwendung finden.

Von den vorgestellten Architekturen ist hier das Cluster als Mischarchitektur im Fokus, da mit dem ebenfalls hier vorgestellten System Postgres-XL eine Clusterlösung zur späteren

Ausführung und Analyse der SQL-Anfragen genutzt wird. Aus den vorgestellten Partitionierungsstrategien ist hier hauptsächlich die Bereichspartitionierung von Bedeutung, da die im Anschluss evaluierten Strategien dieser Art entsprechen. Das Problem der Datenschiefheit findet bei der Erstellung von Partitionierungsstrategien Betrachtung. Die Darstellung des Zwei-Phasen- und Repartitionierungsalgorithmus wird im späteren Verlauf bei der Analyse von Anfrageplänen benötigt. Weniger von Bedeutung sind hier die Lastbalancierung, Replikation und verkettete sowie überlappende Partitionierung. Die Lastbalancierung wird hier von Postgres-XL vorgenommen und ist nicht weiter Bestandteil der Untersuchung. Das Ziel der Replikation und der beiden genannten Partitionierungsarten ist unter anderem die Verbesserung der Ausfallsicherheit, was im Rahmen dieser Arbeit nicht das primäre Ziel ist.

5 Umsetzung

In den vorherigen Kapiteln wurden zunächst typische mathematische Operationen und Algorithmen des wissenschaftlichen Rechnens identifiziert und formal betrachtet. Um diese auf einem normalen und später parallelen Datenbanksystem umzusetzen, ist es im ersten Schritt nötig, eine relationale Darstellung für Matrizen zu finden. Auf Basis dieser ist es anschließend möglich, die Operationen mit der Datenbanksprache SQL zu formulieren. Die gewünschte Parallelisierung dieser Operatoren erfordert eine Aufteilung der Relationen in geeignete Partitionen, so dass die anfallenden Kommunikationskosten möglichst gering sind.

Innerhalb dieses Kapitels ist es daher das Ziel, ein Relationenschema für Matrizen zu bestimmen, SQL-Ausdrücke für die genannten mathematischen Operationen zu formulieren und günstige Partitionierungsstrategien anhand der Matrixmultiplikation zu evaluieren. Abschließend werden die formulierten SQL-Ausdrücke auf dem parallelen Datenbankcluster Postgres-XL ausgeführt und die entstandenen Anfragepläne hinsichtlich der Intraoperatorparallelität untersucht. Dabei ist eine der zu beantwortenden Kernfragen, ob der vorhandene Datenbankoptimierer die Möglichkeiten zur Parallelisierung erkennt und diese in den Anfragepläne umsetzt.

Zur Erreichung dieser Zielstellungen ist dieses Kapitel in fünf Abschnitte unterteilt. In 5.1 wird das Relationenschema vorgestellt, bevor im Abschnitt 5.2 SQL-Anfragen für die mathematische Operationen und in Abschnitt 5.3 für den Vorwärtsalgorithmus formuliert werden. Anschließend erfolgt in Abschnitt 5.4 die Entwicklung von Partitionierungsstrategien, sowie die Berechnung der Kommunikationskosten anhand der Matrixmultiplikation. In Abschnitt 5.5 werden die entwickelten Strategien umgesetzt und hinsichtlich der Anfragepläne und Laufzeiten ausgewertet.

5.1 Matrixdarstellung im Relationenschema

In relationalen Datenbanksystemen, wie zum Beispiel PostgreSQL und Postgres-XL, ist die einzige vorhandene Datenstruktur eine Relation, welche über einem Schema definiert wird. Bei der Ausführung von Operationen oder Anfragen auf einer Relation entsteht immer eine Relation - auch bezeichnet mit dem Kriterium der Abgeschlossenheit. Die Struktur, also das Schema der Ergebnisrelation, ist dabei jedoch abhängig von der Anfrage. Im Rahmen dieser

Arbeit ist es daher notwendig, ein Schema für die Repräsentation einer Matrix zu bestimmen und bei den anschließenden Operationen diese Struktur als Ergebnis zu erzeugen. In [Weu16] wurde ein solches Schema ausführlicher hergeleitet, weshalb hier nur die wesentlichen Punkte aufgeführt werden.

Kriterien für die Matrixrelation sind, dass jedes Element der Matrix eindeutig identifizierbar ist und keine Operation die Integrität der Datenbank verletzten würde. In der einfachsten Form kann eine zweispaltige Matrix als Relation mit zwei Attributen aufgefasst werden.

col1	**col2**
1	2
3	4

Bei dieser Struktur ist jedoch die eindeutige Identifizierbarkeit eines Wertes nicht gegeben. Die Erweiterung um ein Attribut `row`, welche die Tupel und damit Zeilen einer Matrix eindeutig beschreibt, löst dieses Problem. Allerdings wird die zweite Bedingung, dass keine Operation die Integrität verletzt, nicht erfüllt, da beispielsweise bei der Transponierung Attributnamen mit Primärschlüsseln getauscht werden müssten. Die Lösung, welche beide Kriterien erfüllt, besteht aus drei Attributen: `row`, `col` und `val`.

row	**col**	**val**
1	1	1
2	1	3
1	2	2
2	2	4

Jedes Element der Matrix ist bei dieser Struktur eindeutig über die Kombination von `row` und `col` ansprechbar. Darüber hinaus verändert die Transponierung nicht das Datenbankschema. Im Fall dünn besetzter Matrizen können Null-Elemente weggelassen werden.

Im Kommenden wird für Matrizen das hier vorgestellte Schema genutzt. Vektoren werden als Spezialfall aufgefasst und als einspaltige Matrix behandelt.

5.2 Mathematische Operationen in SQL

Nach der Festlegung des Datenbankschema für Matrizen ist es möglich, die in 3.2 vorgestellten, Operationen als SQL-Anfragen zu formulieren. Hierbei geht es zunächst nur darum, Anfragen zu zeigen, die vorgegebene Operationen erfüllen. Die Möglichkeiten zur Parallelisierung werden im Abschnitt 5.4 erläutert und angewendet.

Zur Verdeutlichung der Anfragen seien zwei Matrizen A und B definiert.

$$A = \begin{pmatrix} 5 & 6 & 7 & 5 \\ 3 & 5 & 6 & 8 \\ 3 & 5 & 7 & 4 \\ 7 & 3 & 6 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 3 & 2 \\ 3 & 4 & 5 & 6 \\ 4 & 6 & 6 & 4 \end{pmatrix}$$

Als Relationen `A` und `B` haben die beiden Matrizen folgende Struktur:

Relation `A`

row	col	val
1	1	5
2	1	3
3	1	3
4	1	7
1	2	6
2	2	5
3	2	5
4	2	3
1	3	7
2	3	6
3	3	7
4	3	6
1	4	5
2	4	8
3	4	4
4	4	5

Relation `B`

row	col	val
1	1	1
2	1	2
3	1	3
4	1	4
1	2	2
2	2	3
3	2	4
4	2	6
1	3	3
2	3	3
3	3	5
4	3	6
1	4	4
2	4	2
3	4	6
4	4	4

5.2.1 Skalarprodukt

Das Skalarprodukt zweier Vektoren wird durch $\langle x, y \rangle = x_1 y_1 + ... + x_n y_n$ beschrieben. Hier verwenden wir als Vektoren die erste Spalte $A_{:1}$ der Matrix A und die erste Spalte $B_{:1}$ der Matrix B. Folglich ergibt sich nachstehendes Skalarprodukt.

$$\langle A_{:1}, B_{:,1} \rangle = \begin{pmatrix} 5 \\ 3 \\ 3 \\ 7 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = 5 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 + 7 \cdot 4 = 48$$

Für die Umsetzung mit SQL wurden die beschriebenen Spaltenvektoren in Relationen mit den Bezeichnern `x` und `y` gespeichert. Die Relation `x` entspricht dabei den ersten vier Zeilen von Relation `A` und die Relation `y` den ersten vier Zeilen von Relation `B`. Unter diesen Bedingungen kann das Skalarprodukt mit der Anfrage aus dem Programmbeispiel 5.1 erstellt werden.

```
1 SELECT SUM(x.val * y.val) FROM x JOIN y ON x.row = y.row;
```

Programmbeispiel 5.1: Skalarprodukt mit SQL

Ergebnisrelation

sum
48

Es wird deutlich, dass diese Anfrage auch für dünn besetzte Vektoren funktioniert, da es bei fehlendem Tupel in einer der beiden Relationen keine Übereinstimmung im Verbundattribut gibt.

5.2.2 Matrixmultiplikation

In diesem Abschnitt werden die Matrizen A und B miteinander multipliziert. Dabei wird bereits bei der mathematischen Betrachtung deutlich, dass jeder Zielwert in der entstehenden Matrix das Skalarprodukt eines Zeilen- und Spaltenvektors von A und B ist.

Für die Berechnung der Matrixmultiplikation ist die Anfrage des Skalarproduktes aus 5.1 nur bei den Verbundattributen und der Projektionsliste zu verändern. Statt über die Zeilen zu verknüpfen, wird jetzt `a.col = b.row` gesetzt und zusätzlich über `a.row, b.col` projiziert. Die gesamte Anfrage mit der entsprechenden Ergebnisrelation ist nachfolgend zu sehen.

```
1 SELECT a.row, b.col, SUM(a.val * b.val) FROM a JOIN b ON a.col =
      b.row GROUP BY a.row, b.col;
```

Programmbeispiel 5.2: Matrixmultiplikation mit SQL

Ergebnisrelation

row	col	val
1	1	58
2	1	63
3	1	50
4	1	51
1	2	86
2	2	93
3	2	73
4	2	77
1	3	98
2	3	102
3	3	83
4	3	90
1	4	94
2	4	90
3	4	80
4	4	90

5.2.3 Transponierung

Die Transponierung einer Matrix entspricht dem Vertauschen der Rolle von Zeilen und Spalten. Für die Umsetzung in SQL kann dafür der Umbenennungsoperator genutzt werden, sodass bei der Transponierung von B im `SELECT`-Abschnitt zunächst über die drei Attribute `row`, `col` und `val` projiziert wird, um sie anschließend mit dem `AS`-Operator umzubenennen. Folgende Anfrage transponiert demnach eine Matrix:

```
1 SELECT col as row, row as col, val FROM b;
```

Ergebnisrelation

row	col	val
1	1	1
1	2	2
1	3	3
1	4	4
2	1	2
2	2	3
2	3	4
2	4	6
3	1	3
3	2	3
3	3	5
3	4	6
4	1	4
4	2	2
4	3	6
4	4	4

5.3 Vorwärtsalgorithmus

In diesem Abschnitt wird der Vorwärtsalgorithmus zunächst, wie aus der Definition bekannt, rekursiv und anschließend iterativ umgesetzt. Dabei gibt es für die iterative Umsetzung zwei Varianten: die Nutzung einer verschachtelten Anfrage oder das Überschreiben alter Daten mit der `UPDATE`-Funktion. Die verschachtelte und `UPDATE`-Anfrage sind nicht darauf angewiesen, dass das Datenbanksystem Rekursion unterstützt, weshalb sie auch in Systemen wie MonetDB nutzbar wären. Die hier vorgestellten SQL-Anfragen richten sich nach [MH17].

Im Implementationsteil 5.5 der Arbeit wird der Vorwärtsalgorithmus auf einer großen Menge an Daten getestet, welche zur Visualisierung ungeeignet sind. Daher werden die beiden Anfragen anhand des Beispiels aus der Einführung im Abschnitt 3.3.1 erläutert. Das zu Grunde liegende *HMM* λ hat die Zustände $S = \{Sonne, Regen\}$ und die möglichen Beobachtungen $V = \{nass, trocken\}$. Die Matrizen A und B sowie die Anfangsverteilung π haben, entsprechend des bekannten Relationenschemas, folgende Form.

Relation `A`

row	col	val
1	1	0.8
2	1	0.3
1	2	0.2
2	2	0.7

Relation `B`

row	col	val
1	1	0.1
2	1	0.95
1	2	0.9
2	2	0.05

Relation `Pi`

row	col	val
1	1	0.7
2	1	0.3

Berechnet werden soll hier die Wahrscheinlichkeit für die Beobachtungssequenz $(nass, nass, trocken)$ im beschriebenem *HMM* λ. Als Relation `obs` hat diese folgende Struktur:

Relation `obs`

row	col	val
1	1	1
2	1	1
3	1	2

Die Spalte `val` wird hierbei mit Zahlen gefüllt, wobei 1 für *nass* und 2 für *trocken* steht. Dies ist sinnvoll, da dies gleichzeitig dem Spaltenindex der Matrix B entspricht.

Rekursiv

Vor der Umsetzung des Algorithmus ist es nötig, die Rekursion in SQL zu erklären. PostgreSQL bietet die Möglichkeit, in einer `WITH`-Klausel temporäre Relationen anzulegen, welche nur im Kontext der nachgestellten Anfrage gültig sind. Folgendes Beispiel soll die Syntax verdeutlichen. Die Relation `temp` ist nur in dem nachfolgendem `SFW`-Block gültig.

```
1 WITH temp AS (
2   SELECT row, col, a.val FROM a JOIN b ON a.row = b.row AND a.
      col = b.col
3 ) SELECT row, col, val FROM temp;
```

Mit dem Zusatzmodifikator `RECURSIVE` hinter dem `WITH`-Literal ist es möglich, rekursive Anfragen zu erstellen, da die `WITH`-Anfrage sich auf seinen eigenen Output beziehen kann. Dabei besteht diese Anfrage aus einem nicht-rekursiven Teil, einer Vereinigung mit `UNION` und einem rekursiven Teil, wobei nur im rekursiven Teil die Referenzierung auf den eigenen Output

möglich ist. In der Dokumentation von PostgreSQL [Pos] ist als Einführungsbeispiel folgende Berechnung der Summe der Zahlen von 1 bis 100 aufgeführt.

```
1 WITH RECURSIVE t(n) AS (
2   VALUES (1)
3 UNION ALL
4   SELECT n+1 FROM t WHERE n < 100
5 )
6 SELECT sum(n) FROM t;
```

In der ersten Zeile wird die temporäre Tabelle samt Schema definiert. Die zweite Zeile repräsentiert den nicht rekursiven Teil der Anfrage, welcher über die Vereinigung mit dem rekursiven Teil verknüpft wird.

Bezogen auf die in 3.3.1 vorgestellten Formeln entspricht der Inititalisierungsschritt dem nicht rekursiven Teil, weshalb die Rekursionsformel folglicherweise im zweiten Teil der Anfrage zu beschreiben ist. Daraus ergibt sich für die rekursive Berechnung des Vorwärtsalgorithmus folgender SQL-Abschnitt.

```
1  WITH RECURSIVE alpha (iteration, row, val) AS (
2    SELECT 2, pi.row, (pi.val * b.val)
3      FROM pi JOIN b
4      ON pi.row=b.row and b.col=
5        (SELECT val from obs where i=1)
6  UNION ALL
7    SELECT alp_a.iteration+1, alp_a.row, alp_a.val*b.val
8      FROM b JOIN
9        (SELECT alpha.iteration as iteration, alpha.row as row, a.
             col as col,
10         SUM(alpha.val*a.val) as val
11       FROM alpha JOIN a ON alpha.row=a.row
12       GROUP BY alpha.row, a.col) alp_a
13     ON b.row=alp_a.row
14     WHERE b.col=(SELECT val from obs where obs.row=alp_a.
           iteration)
15     AND alp_a.iteration <= (SELECT max(row) from obs)
```

```
16 ) SELECT SUM(val) FROM alpha WHERE iteration = (SELECT max(
       iteration) from alpha)
```

Programmbeispiel 5.3: Rekursive SQL-Anfrage des Vorwärtsalgorithmus

Obwohl diese Anfrage den Anforderungen des SQL-Standards genügt, ist sie auf aktuellen Versionen von PostgreSQL nicht ausführbar, da dieses System keine Aggregation im rekursiven Teil der Anfrage erlaubt. Beim später verwendeten parallelen Datenbanksystem Postgres-XL ist die Rekursion auf verteilten Relationen gänzlich verboten, weshalb der rekursive Ansatz im Rahmen dieser Arbeit nicht weiter untersucht wird. Da das beschriebene Aggregatfunktionsproblem auch in anderen Systemen, wie MySQL und DB2 [1], auftritt, konnte die Anfrage nicht auf Validität geprüft werden.

Iterativ

Neben der rekursiven Form dieses Algorithmus ist es möglich, iterativ das Ergebnis zu erzeugen. Aus der Rekursionsformel, vorgestellt in Abschnitt 3.3.1, wird deutlich, dass ein Vektor α bei der Ausführung entsteht. Um dies auf SQL-Ebene abzubilden, gibt es zwei Möglichkeiten: Updates auf der Relation `alpha` oder das Nutzen einer verschachtelten Anfrage. Beide Varianten werden nachfolgend am eingeführten Beispiel vorgestellt.

Updatevariante

Analog zur rekursiven Version sind auch hier im Wesentlichen drei Schritte notwendig: Initialisierung, Iteration und Terminierung mit der Summation. Die Initialisierung berechnet mittels der Anfangsverteilung `Pi` und der Matrix `B` die Anfangswerte für `alpha`. In der hier gewählten Implementierung werden in jedem Schritt die bisherigen Werte überschrieben, anstatt weitere anzufügen, weshalb ein zweizeiliger Vektor entsteht. Dies ist möglich, da nur die letzten Werte von `alpha` im Terminierungsschritt summiert werden. Mit SQL lässt sich das folgendermaßen realisieren:

```
1 INSERT INTO alpha
2   SELECT Pi.row, 1, Pi.val * B.val FROM
3     Pi JOIN B ON Pi.row = B.row
4     WHERE B.col = (SELECT val FROM obs WHERE row = 1 AND col =
        1);
```

Programmbeispiel 5.4: Initialisierung Updatevariante

[1] Erkenntnis von Projektgruppen des Moduls *Neueste Entwicklungen in der Informatik* im Sommersemester 2017.

Nach dieser Anfrage ergibt sich folgender Zwischenstand.

Relation `alpha`

row	col	val
1	1	0.07
2	1	0.285

Auf diesem Stand kann die Iteration über die weiteren Beobachtungszeitpunkte ansetzen. Dabei werden bei jedem Schritt die Werte von `alpha` mit einem `UPDATE` erneuert. Hier sind aufgrund der Beobachtungssequenzlänge von drei insgesamt zwei `UPDATE`-Aufrufe nötig.

```
1  -- Zeitpunkt t = 2
2  UPDATE alpha SET val = (
3    SELECT alpha_a.val * B.val FROM
4      (SELECT A.col as row, SUM(alpha.val * A.val) as val FROM
          alpha
5      JOIN A ON alpha.row = A.row GROUP BY A.col) alpha_a
6    JOIN B ON alpha_a.row = B.row
7    WHERE B.col = (SELECT val FROM obs WHERE row = 2 and col = 1)
       AND B.row = alpha.row)
8    WHERE EXISTS (SELECT * FROM (SELECT DISTINCT col as row FROM A
       ) ttt WHERE ttt.row = alpha.row);
9
10 -- Zeitpunkt t = 3
11 UPDATE alpha SET val = (
12   SELECT alpha_a.val * B.val FROM
13     (SELECT A.col as row, SUM(alpha.val * A.val) as val FROM
          alpha
14     JOIN A ON alpha.row = A.row GROUP BY A.col) alpha_a
15   JOIN B ON alpha_a.row = B.row
16   WHERE B.col = (SELECT val FROM obs WHERE row = 3 and col = 1)
       AND B.row = alpha.row)
17   WHERE EXISTS (SELECT * FROM (SELECT DISTINCT col as row FROM A
       ) ttt WHERE ttt.row = alpha.row);
```

Programmbeispiel 5.5: Update der Relation alpha

Relation `alpha`

row	col	val
1	1	0.06495075
2	1	0.007240375

Der letzte Schritt erfordert die Summation über die `val`-Spalte des Vektors `alpha`.

```
1 SELECT SUM(val) FROM alpha;
```

Programmbeispiel 5.6: Summation der val-Spalte von alpha

`SUM(val)`

sum
0.072191125

Die Wahrscheinlichkeit für die Beobachtungssequenz $(nass, nass, trocken)$ im beschriebenen HMM λ ist 7.2%. Die gleiche Berechnung soll nachfolgend mit einer verschachtelten Variante durchgeführt werden.

Verschachtelte Variante

Vor der Darstellung der gesamten Anfrage werden die Komponenten dieser einzeln vorgestellt. Auch hier ist wieder ein Initialisierungs-, Iterations- und Terminierungsabschnitt vorhanden. Die einzelnen Anfragen werden dann im letzten Schritt zu einer Gesamtanfrage zusammengesetzt.

Die Initialisierung ist dieselbe Anfrage wie bei der Update-Variante, mit dem Unterschied, dass das Ergebnis nicht in die Relation `alpha` geschrieben wird. Diese entspricht derer zum Zeitpunkt $t = 1$ und ist die innerste Subanfrage.

```
1 SELECT Pi.row as row, Pi.val * B.val as val FROM PI JOIN B on Pi
    .row = B.row WHERE B.col = (SELECT val FROM obs WHERE row = 1
    AND col = 1)
```

Programmbeispiel 5.7: Initialisierung bei geschachtelter Anfrage

Die weiteren Anfragen gelten für alle Zeitpunkte $t > 1$ und beinhalten gleichzeitig die vorherigen Subanfragen, welche hier mit `Q(t-1)` gekennzeichnet sind. In der letzten Zeile entspricht der Buchstabe `t` keinem Attribut, sondern der Variable t.

```
SELECT alpha_a.row as row, alpha_a.val * B.val as val FROM (
   SELECT A.col as row, SUM(aa.val * A.val) as val FROM Q(t-1) as
    aa JOIN A ON aa.row = A.row GROUP BY A.col) alpha_a JOIN B on
    alpha_a.row = B.row WHERE b.col = (SELECT val FROM obs WHERE
   row = t and col = 1);
```

Programmbeispiel 5.8: Iterationsanfrage

Im Anschluss ist es möglich, über das temporäre Ergebnis der geschachtelten Anfrage zum Zeitpunkt T zu aggregieren.

```
SELECT SUM(val) FROM Q(T)
```

Programmbeispiel 5.9: Summation der geschachtelten Anfrage

Zum Abschluss wird hier die gesamte Anfrage, welche sich aus den beschriebenen Komponenten zusammensetzt und die Wahrscheinlichkeit der Beobachtungssequenz $(nass, nass, trocken)$ berechnet, gezeigt.

```
-- Start Summation
SELECT SUM(val) FROM
  -- Start Subanfrage t=3
  (SELECT alpha_a.row as row, alpha_a.val * B.val as val FROM (
     SELECT A.col as row, SUM(aa.val * A.val) as val FROM
    -- Start Subanfrage t=2
    (SELECT alpha_a.row as row, alpha_a.val * B.val AS val FROM
       (SELECT A.col as row, SUM(aa.val * A.val) as val FROM
      -- Start Subanfrage t=1
      (SELECT Pi.row as row, Pi.val * B.val as val FROM PI JOIN
         B on Pi.row = B.row WHERE B.col = (SELECT val FROM obs
         WHERE row = 1 AND col = 1)) as aa
      -- Ende Subanfrage t=1
```

```
10      JOIN A ON aa.row = A.row GROUP BY A.col) alpha_a JOIN B on
           alpha_a.row = B.row WHERE B.col = (SELECT val FROM obs
           WHERE row = 2 AND col = 1)) as aa
11      -- Ende Subanfrage t=2
12    JOIN A ON aa.row = A.row GROUP BY A.col) alpha_a JOIN B on
         alpha_a.row = B.row WHERE b.col = (SELECT val FROM obs WHERE
          row = 3 and col = 1))
13    -- Ende Subanfrage t=3
14  as complete_query;
15  -- Ende Summation
```

Programmbeispiel 5.10: Geschachtelte Anfrage des Vorwärtsalgorithmus

SUM(val)

sum
0.072191125

Bei der Ausführung führt diese Anfrage erwartungsgemäß zum gleichen Ergebnis wie die vorherige Variante. Es ist davon auszugehen, dass die `UPDATE`-Variante eine schlechtere Performance hat, da bei jedem `UPDATE` die Transaktionssicherheit gewährleistet sein muss. Desweiteren wird bei jeder Anfrage persistent in die Datenbank geschrieben, was zu einer höheren Belastung führt. Aus Zeitgründen wurde diese Vermutung nicht weiter untersucht.

5.4 Parallelisierung der SQL-Anfragen

In den bisherigen Abschnitten dieses Kapitels wurden ein Relationenschema für Matrizen festgelegt und darauf aufbauende SQL-Anfragen für die Matrixmultiplikation und den Vorwärtsalgorithmus formuliert.

Um eine Basis für die angestrebte Intraoperatorparallelität zu schaffen, ist nun die Partitionierung von Relationen zu untersuchen. Mögliche Strategien werden daher im Abschnitt 5.4.1 vorgestellt. Da bei verteilten Systemen die Kommunikationskosten oftmals den Verarbeitungsprozess dominieren, erfolgt im Abschnitt 5.4.2 die Ermittlung dieser für die einzelnen Partitionierungsvarianten. Dieses Vorgehen soll dem Datenbanksystem möglichst gute Voraussetzungen für die Umsetzung der Intraoperatorparallelisierung schaffen. Die Analyse, ob der Anfrageplanoptimierer des Systems die vorhandenen Potenziale ausnutzt, findet im Abschnitt 5.5 statt.

5.4.1 Partitionierungsstrategien

Die Zerlegung einer Relation in kleinere Bestandteile, mit der anschließenden Verteilung auf verschiedene Knoten, ist eine wesentliche Voraussetzung der Intraoperatorparallelität, da nur so Operatorinstanzen auf einzelnen, kleineren Abschnitten agieren können. Eine Verteilung und Replikation ganzer Relationen würde nur Interanfrage- bzw. Interoperatorparallelität ermöglichen. In diesem Abschnitt soll auf Basis der in 4.3.1 vorgestellten Strategien eine günstige, anhand nachfolgender Kriterien, ermittelt werden.

Im Rahmen dieser Arbeit sollen Operationen des maschinellen Lernens betrachtet werden. Aus den Einführungen wird bereits deutlich, dass bei solchen Algorithmen Multiplikationen zwischen Matrizen und Vektoren eine große Bedeutung haben. Ziel ist es daher, eine Partitionierungsform zu wählen, die für diese Operationen vorteilhaft ist. Hier kann angenommen werden, dass eine Strategie gut ist, wenn mit ihr wenig Kommunikation nötig ist. Dies ist dann möglich, wenn viele der Datenabschnitte bereits auf dem jeweiligen Knoten vorhanden sind. Als weiteres Kriterium ist wichtig, dass die einzelnen Knoten möglichst gleich viele Daten bearbeiten. Dieses Problem wurde in 4.4.2 als Datenschiefheit eingeführt. Darüber hinaus soll die gewählte Strategie gegenüber Veränderungen möglichst robust sein. Dazu zählt zum Beispiel das Vertauschen der Matrizen bei der Matrixmultiplikation oder gar die Verwendung einer komplett anderen Operation.

Zur Visualisierung werden die Matrizen A und B aus Abschnitt 5.2 auf zwei Knoten verteilt. Die vorgestellten Varianten sind zeilenweise, spaltenweise, zeilen-spaltenweise und Blockpartitionierung.

Zeilenweise Partitionierung

Bei der hier vorgestellten zeilenweisen Partitionierung werden alle Elemente einer Zeile einem Knoten zugeordnet. Die Knotennummer $s(i)$ der entsprechenden Zeile i lässt sich über $s(i) = (i - 1 \mod m)$ bestimmen. Andere Formen, wie mehrere Zeilen hintereinander einem Knoten zuzuordnen, werden hier nicht betrachtet.

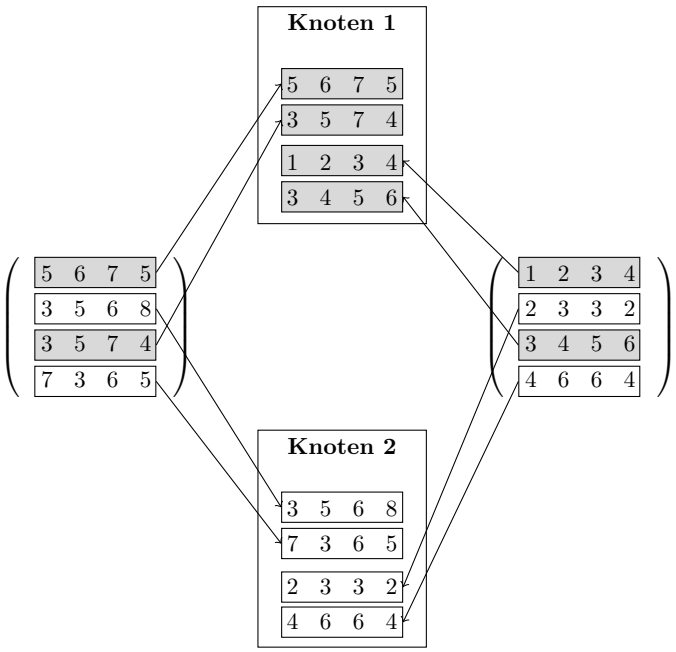

Abbildung 5.1: Zeilenweise Partitionierung

Spaltenweise Partitionierung

Analog zur zeilenweisen Partitionierung werden hier alle Elemente einer Spalte einem Knoten zugeordnet. Dementsprechend ergibt sich die Knotennummer $s(i)$ der Spalte j über $s(j) = (j-1 \mod m)$.

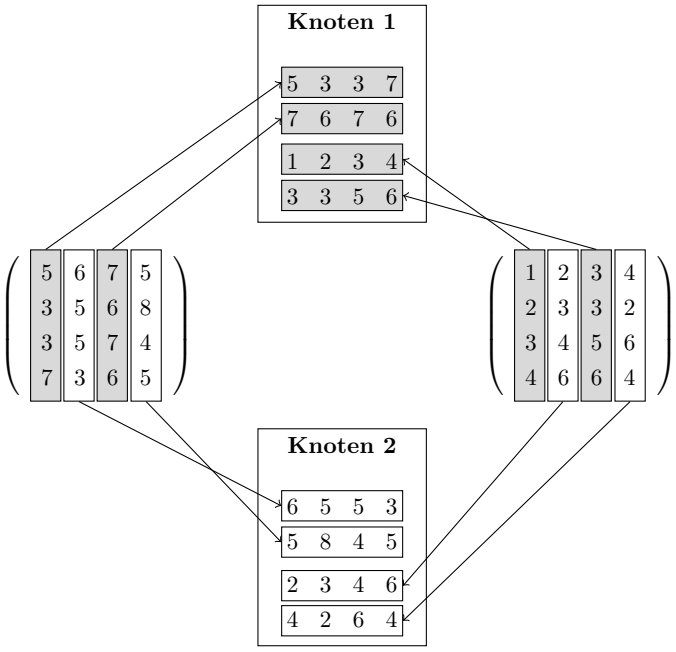

Abbildung 5.2: Spaltenweise Partitionierung

Zeilen-Spaltenweise Partitionierung

Hierbei handelt es sich um eine Kombination aus den beiden vorherigen Partitionierungsarten mit dem Ziel, die Eigenschaft der Matrixmultiplikation, das Skalarprodukt zwischen Zeilen und Spalten zu bilden, zu nutzen. Dafür wird die Matrix A zeilenweise und die Matrix B spaltenweise partitioniert.

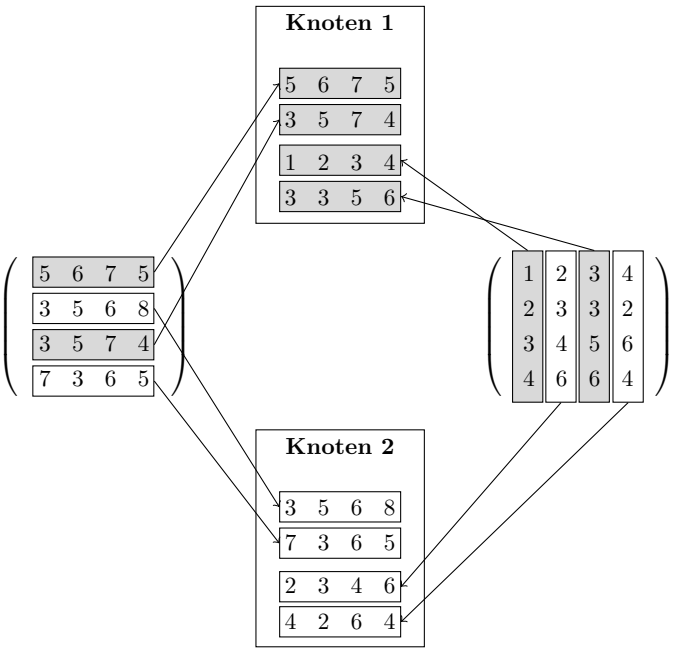

Abbildung 5.3: Zeilen-Spaltenweise Partitionierung

Blockweise

Im Abschnitt 3.2.2 und 4.3.1 wurde bereits das Verhalten der Matrixmultiplikation bei Blockmatrizen beschrieben. Um dieses anwenden zu können, ist es sinnvoll, die Matrizen in Blöcke zu teilen. Bei dieser Art ist es erneut möglich, zwischen Zeilenweise, Spaltenweise oder gemischt zu unterscheiden. Unabhängig davon besteht im hier gewählten Beispiel ein Block aus $x \cdot n$ Elementen.

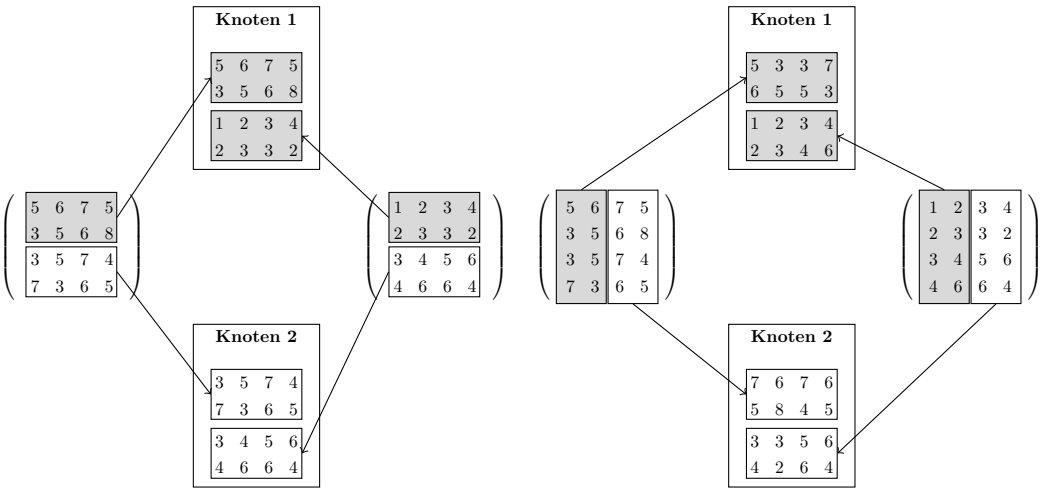

Abbildung 5.4: Blockweise Partitionierung

Im folgenden Abschnitt wird für die vorgestellten Partitionsstrategien eine Berechnung der Kommunikationskosten vorgenommen.

5.4.2 Kommunikationskosten

Die Berechnung der Matrixmultiplikation auf mehreren Knoten erfordert immer das Versenden von Elementen zwischen diesen. Die dadurch entstehenden Kommunikationskosten sind ein bekannter Flaschenhals vieler verteilter Berechnungen und sollen deshalb möglichst gering sein. Demzufolge ist es gut, viele Berechnungen bereits ohne Senden realisieren zu können. Bevor hier für die im Abschnitt 5.4.1 vorgestellten Strategien Kommunikationskosten und lokal berechenbare Anteile ermittelt werden, ist es sinnvoll, die Berechnungsvorschrift der Matrixmultiplikation, hier für $A \cdot B = C$, zu untersuchen.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{pmatrix} =$$

$$\begin{pmatrix} \sum_{k=1}^{n} a_{1,k}b_{k,1} & \sum_{k=1}^{n} a_{1,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{1,k}b_{k,n} \\ \sum_{k=1}^{n} a_{2,k}b_{k,1} & \sum_{k=1}^{n} a_{2,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{2,k}b_{k,n} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{k=1}^{n} a_{n,k}b_{k,1} & \sum_{k=1}^{n} a_{n,k}b_{k,2} & \cdots & \sum_{k=1}^{n} a_{n,k}b_{k,n} \end{pmatrix}$$

Aus dieser Aufstellung wird zunächst deutlich, dass für die Ergebnismatrix $n \times n$ Zielelemente berechnet werden müssen. Für die Berechnung eines dieser Elemente von C gilt

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \cdots + a_{i,n} b_{n,j}.$$

Es werden demnach $2n$ Elemente für die Berechnung eines $c_{i,j}$ benötigt, weshalb für die gesamte Matrix C $2n^3$ Elemente agieren. Die letzte Formel verdeutlicht ebenfalls eine Zweiteilung in Multiplikation und Summation. Die nachfolgende Kostenermittlung wird deshalb in zwei Phasen unterteilt:

1. Multiplikationphase
2. Summationsphase

In der Multiplikationsphase werden die Elemente betrachtet, welche für die Multiplikation notwendig sind. Die zur Bildung der gesamten Summe anfallenden Kosten werden der zweiten Phase zugeordnet.

Die Ermittlung der Kommunikationskosten erfolgt auf Matrizen mit einer Dimension von $n \times n$, die auf eine Knotenanzahl m verteilt werden. Darüber hinaus soll n ein Vielfaches der Knotenanzahl von m sein. Die Einschränkung auf $n \times n$ Matrizen ist im Bereich der Sensordatenverarbeitung beispielsweise über eine Verkleinerung des Datensatzes auf das nächst mögliche n zu realisieren, da bei der vorhandenen Datengröße durch diese keine Aussageänderung zu erwarten ist. Auch in weiteren Algorithmen sind quadratische Matrizen Bestandteil oder Grundlage, wie beispielsweise die Übergangswahrscheinlichkeitsmatrix A des HMM (Abschnitt 3.3) und die Matrix bei der Berechnung des Google PageRank [LM03].

Zusammenfassend werden die beschriebenen Bedingungen aufgeführt:

- Matrixdimension: $n \times n$, wobei $n = x \cdot m$ mit $x \in \mathbb{Z}$
- Knotenanzahl: m, es gilt $m = n/x$
- Vielfaches: x, es gilt $x = n/m$

Mit Bezug auf die genannten Bedingungen ergibt sich eine Aussage über die Kommunikationskosten, welche für alle Strategien gilt: Jede Zeile benötigt jede Spalte. Das bedeutet für n Zeilen und n Spalten mit wiederum n Elementen sind die Maximalkosten n^3. Bei einem Vielfachen $x > 1$ verwaltet ein Knoten jedoch mehrere Zeilen, wodurch nicht an n Zeilen,

sondern m Knoten gesendet werden muss. Demnach ist die obere Begrenzung der Kosten in der Multiplikationsphase

$$n^2 m. \tag{5.1}$$

In den kommenden Kostenberechnungen wird daher ermittelt, welche Elemente bereits lokal vorhanden sind, um sie von $n^2 m$ abzuziehen. Für die Blockpartitionierung kann diese Formel abgewandelt werden, was im Abschnitt 5.4.2 erläutert wird. Die Kosten aus Formel 5.1 entstehen nur, wenn die beiden Matrizen auf disjunkten Mengen von Knoten gespeichert sind.

Die nachfolgenden Bereiche gliedern sich jeweils in einen allgemeinen Teil, in dem die Formeln zur Berechnung der Kosten Betrachtung finden und einen Veranschaulichungsteil, in dem einzelne Kommunikationsabschnitte anhand der Beispielmatrizen gezeigt werden.

Kosten bei zeilenweise Partitionierung

Im Rahmen aller Kostenberechnungen soll zur Einstiegsvisualisierung die Darstellung einer Multiplikation zweier 4×4 Matrizen dienen, bei denen die Zahlen den Knotennummern entsprechen, auf dem das jeweilige Element liegt. Innerhalb der Ergebnismatrix beschreibt das Tupel $(1, 2)$ eine Multiplikation zweier Elemente, von denen das Erste auf Knoten 1 und das Zweite auf Knoten 2 liegt. Die einzelnen Tupel, die bei der Summation benötigt werden, sind durch Kommas getrennt. Bei dieser Partitionierungsstrategie ergibt sich folgendes Schema.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix} =$$

$$\begin{pmatrix} \textbf{(1,1)}, (1,2), (1,3), (1,4) & \textbf{(1,1)}, (1,2), (1,3), (1,4) & \textbf{(1,1)}, (1,2), (1,3), (1,4) & \textbf{(1,1)}, (1,2), (1,3), (1,4) \\ (2,1), \textbf{(2,2)}, (2,3), (2,4) & (2,1), \textbf{(2,2)}, (2,3), (2,4) & (2,1), \textbf{(2,2)}, (2,3), (2,4) & (2,1), \textbf{(2,2)}, (2,3), (2,4) \\ (3,1), (3,2), \textbf{(3,3)}, (3,4) & (3,1), (3,2), \textbf{(3,3)}, (3,4) & (3,1), (3,2), \textbf{(3,3)}, (3,4) & (3,1), (3,2), \textbf{(3,3)}, (3,4) \\ (4,1), (4,2), (4,3), \textbf{(4,4)} & (4,1), (4,2), (4,3), \textbf{(4,4)} & (4,1), (4,2), (4,3), \textbf{(4,4)} & (4,1), (4,2), (4,3), \textbf{(4,4)} \end{pmatrix}$$

Es wird deutlich, dass nur in der ersten Phase Elemente gesendet werden müssen, da nach dem Senden der Spaltenelemente alle zur Summation nötigen Multiplikationspaare auf einem Knoten liegen. Darüber hinaus ist mindestens eines der Faktorenpaare lokal vorhanden. Da für dieses nicht gesendet werden muss, verringert sich die Anzahl aus Formel 5.1. Die Menge dieser Paare pro Ergebniselement ist abhängig vom Vielfachen x, welches wiederum als $\frac{n}{m}$ darstellbar ist. Diese Anzahl an Paaren ist für m Knoten und n Spalten in Formel 5.1 enthalten und muss demnach abgezogen werden. Über Umformungen von $m \cdot n \cdot \frac{n}{m}$ ergibt sich n^2. Wie eingangs beschrieben, fallen in der zweiten Phase keine Kosten an, wodurch sich die Gesamtkosten wie folgt aufschlüsseln:

- Multiplikationsphase: $n^2m - n^2$ zu versendende Elemente
- Summationsphase: keine zu versendenden Elemente

Der Sendevorgang soll anhand der bekannten Beispielmatrizen verdeutlicht werden. Ausgehend davon, dass jede Berechnung eines Zielelementes gleich abläuft, wird hier die des Elementes $c_{1,3}$ betrachtet.

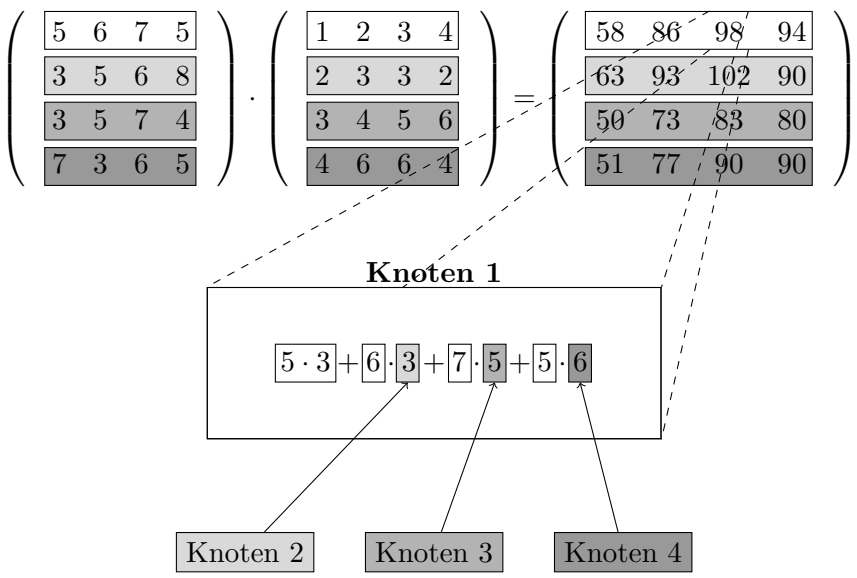

Abbildung 5.5: Berechnung des Elementes $c_{1,3}$ bei Zeilenpartitionierung

Kosten bei spaltenweise Partitionierung

Bei dieser Partitionsstrategie ist analog zur Zeilenweise-Partitionierung nur das Senden von Elementen in der ersten Phase notwendig, in der erneut Faktorenpaare lokal zur Verfügung stehen.

Zunächst wird hier wieder, anhand des im letzten Abschnitt vorgestellten Schemas, die Belegung der Knoten und ihre Bedeutung innerhalb der Zielmatrix dargestellt.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} =$$

$$\begin{pmatrix} \mathbf{(1,1)},(2,1),(3,1),(4,1) & (1,2),\mathbf{(2,2)},(3,2),(4,2) & (1,3),(2,3),\mathbf{(3,3)},(4,3) & (1,4),(2,4),(3,4),\mathbf{(4,4)} \\ \mathbf{(1,1)},(2,1),(3,1),(4,1) & (1,2),\mathbf{(2,2)},(3,2),(4,2) & (1,3),(2,3),\mathbf{(3,3)},(4,3) & (1,4),(2,4),(3,4),\mathbf{(4,4)} \\ \mathbf{(1,1)},(2,1),(3,1),(4,1) & (1,2),\mathbf{(2,2)},(3,2),(4,2) & (1,3),(2,3),\mathbf{(3,3)},(4,3) & (1,4),(2,4),(3,4),\mathbf{(4,4)} \\ \mathbf{(1,1)},(2,1),(3,1),(4,1) & (1,2),\mathbf{(2,2)},(3,2),(4,2) & (1,3),(2,3),\mathbf{(3,3)},(4,3) & (1,4),(2,4),(3,4),\mathbf{(4,4)} \end{pmatrix}$$

Analog zur Zeilenpartitionierung sind für jedes Ergebniselement Faktorenpaare lokal vorhanden. Um wieder nur in der ersten Phase Kommunikationskosten zu haben, ist es sinnvoll, alle Zeilen der linken Matrix an die Spaltenknoten der rechten Matrix zu senden. Die Formel 5.1 bleibt davon unberührt. Die Anzahl der in dieser zu viel bedachten Paare ist wiederum mit n^2 ermittelbar, wodurch sich die gleichen Gesamtkosten wie bei der Zeilenpartitionierung ergeben.

Kostenübersicht Spaltenpartitionierung:

- Multiplikationsphase: $n^2m - n^2$ zu versendende Elemente
- Summationsphase: keine zu versendenden Elemente

Auch hier wird die Berechnung des Elementes $c_{1,3}$ an den bekannten Matrizen visualisiert.

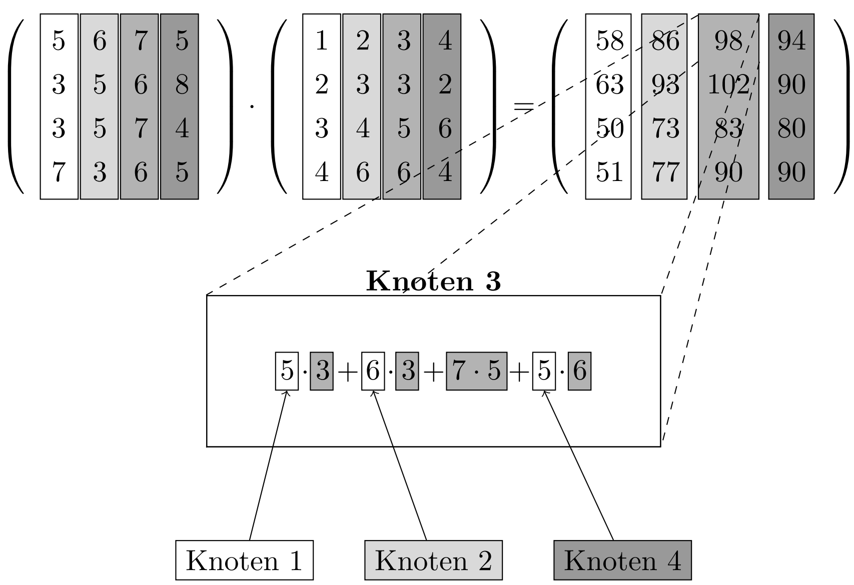

Abbildung 5.6: Berechnung des Elementes $c_{1,3}$ bei Spaltenpartitionierung

Kosten bei zeilen-spaltenweise Partitionierung

Die Berechnungsvorschrift der Matrixmultiplikation, bei der jeweils Zeilen der ersten Matrix mit Spalten der zweiten Matrix verknüpft werden, legt eine unterschiedliche Verteilung der beiden Matrizen nahe. Die Aufteilung und Berechnung der einzelnen Elemente dieser Partitionierung geschieht schematisch wie folgt:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} =$$

$$\begin{pmatrix} \mathbf{(1,1),(1,1),(1,1),(1,1)} & (1,2),(1,2),(1,2),(1,2) & (1,3),(1,3),(1,3),(1,3) & (1,4),(1,4),(1,4),(1,4) \\ (2,1),(2,1),(2,1),(2,1) & \mathbf{(2,2),(2,2),(2,2),(2,2)} & (2,3),(2,3),(2,3),(2,3) & (2,4),(2,4),(2,4),(2,4) \\ (3,1),(3,1),(3,1),(3,1) & (3,2),(3,2),(3,2),(3,2) & \mathbf{(3,3),(3,3),(3,3),(3,3)} & (3,4),(3,4),(3,4),(3,4) \\ (4,1),(4,1),(4,1),(4,1) & (4,2),(4,2),(4,2),(4,2) & (4,3),(4,3),(4,3),(4,3) & \mathbf{(4,4),(4,4),(4,4),(4,4)} \end{pmatrix}$$

Aus der Übersicht wird deutlich, dass für die n Diagonalelemente keine Kommunikation nötig ist. Für die restlichen $n^2 - n$ Elemente ist jedoch keines der Faktorenpaare auf einem gleichen Knoten, weshalb $(n^2 - n) \cdot (m - 1)$ Kosten anfallen. Sollte das Vielfache x jedoch größer 1 sein, so lassen sich die Kosten mit $n^2 \cdot (m - 1)$ errechnen, da $m - 1$ Knoten n-mal-n Spaltenelemente benötigen. Ausklammern dieser Formel führt zur bekannten Darstellung $n^2 m - n^2$. Diese gilt unabhängig davon, ob die Zielmatrix Zeilen- oder Spaltenform hat. In der zweiten Phase entstehen wiederum keine Kosten.

Es lässt sich zusammenfassen:

- Multiplikationsphase: $n^2 m - n^2$ zu versendende Elemente
- Summationsphase: keine zu versendenden Elemente

Da sich hier die Bestimmung der Diagonalelemente von den Anderen unterscheidet, werden in der nachfolgenden Abbildung stellvertretend $c_{2,2}$ und $c_{1,3}$ betrachtet.

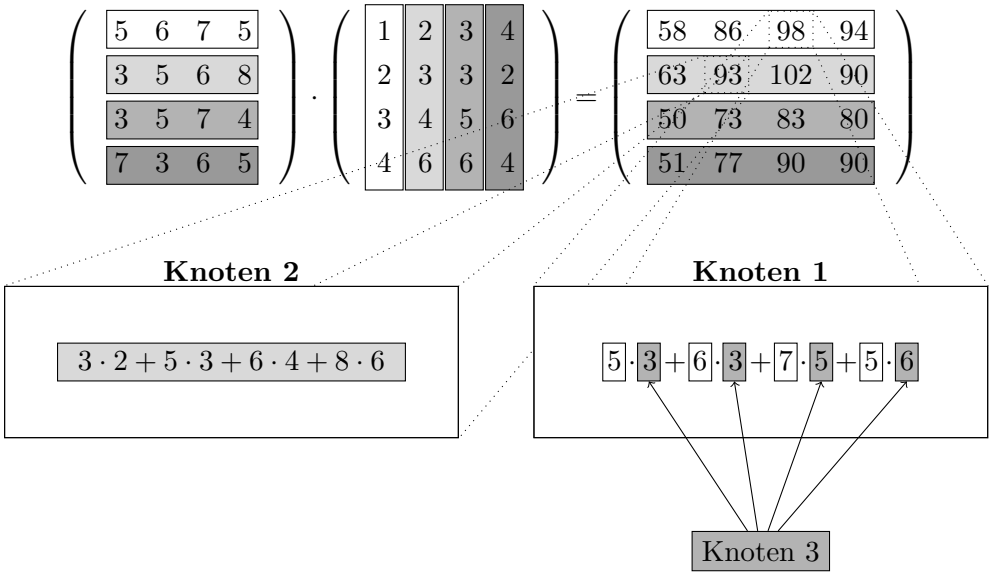

Abbildung 5.7: Berechnung der Elemente $c_{2,2}$ und $c_{1,3}$ bei Zeilen-Spaltenpartitionierung

Kosten für Blockpartitionierung

Für die Betrachtung der Kosten dieser Strategie wird die oben angeführte Bedingung hinsichtlich der Knotenanzahl eingeschränkt, um eine Partitionierung in quadratische Blöcke zu ermöglichen. Hierfür muss die Knotenanzahl m eine Quadratzahl k^2, $k \in \mathbb{Z}$ sein. Für die Dimension n der Matrix in diesem Abschnitt gilt: $n = x \cdot \sqrt{m} = x \cdot k$. Diese Variante ermöglicht zum einen die in 3.2.2 vorgestellte Berechnungsvorschrift für Blockmatrizen anzuwenden. Zum anderen ist aber auch von gesteigerter Robustheit gegenüber veränderten Operationen auszugehen.

Die definierte Knotenlandschaft kann logisch als quadratisches Netz mit den Abmaßen $k \times k$ angesehen werden, auf das die Matrizen der Dimension $(x \cdot k) \times (x \cdot k)$ so verteilt werden, dass die Blöcke selbst wieder quadratisch sind. Da in diesem quadratischen Netz k Knoten pro Zeile und Spalte liegen, ist die Größe für einen Block folglich auf $x \times x$ festzulegen.

Die Kostenberechnung wird hier einmal anhand der standardmäßigen Berechnungsvorschrift und anschließend auf Basis derer für Blockmatrizen (beide Fälle in 3.2.2 beschrieben) durchgeführt. Bei beiden ist ein Versenden von Elementen in der Multiplikations- und Summationsphase nötig.

Die eingeführte Formel 5.1 zur Maximalkostenberechnung in der Multiplikationsphase muss hier angepasst werden. Dort wurde angenommen, dass eine Spalte nur einmal zu einem Knoten geschickt werden muss, wenn dieser mehr als eine Zeile speichert, weshalb mit der Gesamtzahl aller Spaltenelemente[2] n^2 und der Anzahl Knoten m die obere Schranke der Kosten errechenbar waren. Für die Blockpartitionierung kann statt m die Anzahl der Blockzeilen, ausgedrückt mit k, angenommen werden. Die Formel lautet nun also

$$n^2 k. \tag{5.2}$$

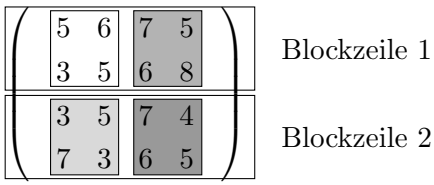

Abbildung 5.8: Blockzeilen einer Matrix

[2] Jedes Element einer Matrix ist ein Spalten- und Zeilenelement

Konventionelle Berechnung Bei der konventionellen Matrixmultiplikation ergeben sich in der Multiplikationsphase blockweise unterschiedliche Fälle: Zielelementberechnungen ohne lokal vorhandene Faktorenpaare und mit lokal vorhandenen Faktorenpaaren. Zur Verdeutlichung wird wieder die bekannte Darstellung genutzt.

$$\begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{pmatrix} =$$

$$\begin{pmatrix} \mathbf{(1,1),(1,1)}, (2,3), (2,3) & \mathbf{(1,1),(1,1)}, (2,3), (2,3) & (1,2), (1,2), (2,4), (2,4) & (1,2), (1,2), (2,4), (2,4) \\ \mathbf{(1,1),(1,1)}, (2,3), (2,3) & \mathbf{(1,1),(1,1)}, (2,3), (2,3) & (1,2), (1,2), (2,4), (2,4) & (1,2), (1,2), (2,4), (2,4) \\ (3,1), (3,1), (4,3), (4,3) & (3,1), (3,1), (4,3), (4,3) & (3,2), (3,2), \mathbf{(4,4),(4,4)} & (3,2), (3,2), \mathbf{(4,4),(4,4)} \\ (3,1), (3,1), (4,3), (4,3) & (3,1), (3,1), (4,3), (4,3) & (3,2), (3,2), \mathbf{(4,4),(4,4)} & (3,2), (3,2), \mathbf{(4,4),(4,4)} \end{pmatrix}$$

In der Beschreibung von Formel 5.2 wurde bereits erläutert, dass eine Spalte nur einmal pro Block versendet werden muss. Jedoch sind in dieser Formel, wie auch in Formel 5.1, die lokal vorhandenen Elemente mitbetrachtet, welche daher identifiziert und anschließend abgezogen werden müssen. Im Allgemeinen bezieht sich diese Begebenheit auf die Elemente der Diagonalblöcke einer Ergebnismatrix, welche im Beispiel die Blöcke links oben und rechts unten sind. Wird diese Situation aus Sicht der gesendeten Spalten betrachtet, so ist zu erkennen, dass x Elemente jeder Spalte in einer der k Blockzeilen überflüssig sind. Bezogen auf das Beispiel sind $c_{1,1}$, $c_{1,2}$, $c_{2,1}$, $c_{2,2}$ für die ersten beiden Spalten, sowie $c_{3,3}$, $c_{3,4}$, $c_{4,3}$ und $c_{4,4}$ für die letzten beiden Spalten, bereits auf dem richtigen Knoten. Demnach ist von der Formel 5.2 die Anzahl der Spalten n mal das Vielfache x abzuziehen.

Die Kosten der Multiplikationsphase sind daher ermittelbar über folgende Formel:

$$n^2k - xn$$

Wie bereits beschrieben, werden hier in beiden Phasen Elemente versendet. In der Zweiten, der Summationsphase, werden jene betrachtet, die zum Errechnen der Gesamtsumme nötig sind. Im Allgemeinen sind das die Ergebnisse der einzelnen Multiplikationen, wovon pro Zielelement n existieren. Da jedoch jeweils n/k Multiplikationen auf dem gleichen Knoten stattfinden, ist es möglich auf diesen bereits Teilsummen zu bilden. Die Anzahl der für die Gesamtsummation benötigten Elemente verringert sich dadurch auf k und es müssen $k-1$ zu einem anderen

Knoten geschickt werden. Die Summation erfordert somit das Versenden von

$$n^2 \cdot (k-1) = n^2k - n^2$$

Teilsummen.

Die Addition der einzelnen Kosten ergibt die Gesamtkosten:

$$n^2k - xn + n^2k - n^2$$

Im Folgenden sollen die aufgeführten Fälle und Berechnungen mittels einer Grafik verdeutlicht werden, wobei hier beispielhaft ein Diagonalelement und ein anderes Element Betrachtung findet.

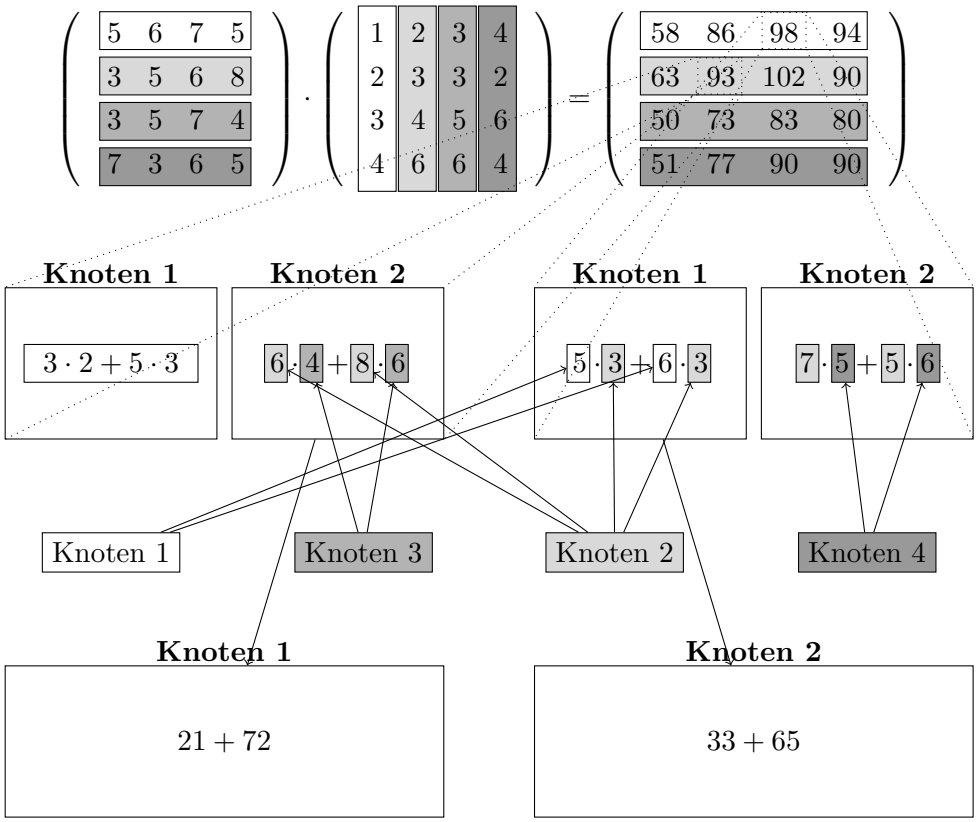

Abbildung 5.9: Berechnung der Elemente $c_{2,2}$ und $c_{1,3}$ bei Blockpartitionierung

Blockmatrixberechung Wie bereits erwähnt ist ein Vorteil der Blockpartitionierung die Anwendbarkeit der in 3.2.2 vorgestellten Berechnungsvorschrift für Blockmatrizen. Die Herleitung soll hier anhand der Beispielmatrizen aus 5.2 erfolgen.

Beide Beispielmatrizen lassen sich in vier Submatrizen aufteilen, die wiederum auf vier Knoten verteilt werden. Die Indizes von A und B entsprechen den Knotennummern.

$$\begin{bmatrix} \begin{pmatrix} 5 & 6 \\ 3 & 5 \end{pmatrix} & \begin{pmatrix} 7 & 5 \\ 6 & 8 \end{pmatrix} \\ \begin{pmatrix} 3 & 5 \\ 7 & 3 \end{pmatrix} & \begin{pmatrix} 7 & 4 \\ 6 & 5 \end{pmatrix} \end{bmatrix} \cdot \begin{bmatrix} \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} & \begin{pmatrix} 3 & 4 \\ 4 & 6 \end{pmatrix} \\ \begin{pmatrix} 3 & 4 \\ 3 & 2 \end{pmatrix} & \begin{pmatrix} 5 & 6 \\ 6 & 4 \end{pmatrix} \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \cdot \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix}$$

Da hier einzelne Blöcke betrachtet werden, ist es möglich, die Formel 5.2 anzupassen: An k Blockzeilen sind k Blockspalten mit k Blöcken zu senden, wodurch sich folgende Formel ergibt:

$$k^3 \tag{5.3}$$

Analog zur konventionellen Variante werden hier in der Diagonale Blöcke mitbetrachtet, die bereits lokal vorhanden sind. Pro versendeter Blockspalte betrifft dies einen der k Spaltenblöcke, weshalb insgesamt $k \cdot 1 = k$ Blöcke von Formel 5.3 abgezogen werden müssen. Da ein Block aus x^2 Elementen besteht, sind die Kosten der Multiplikationsphase ausdrückbar mit:

$$x^2k^3 - x^2k = n^2k - xn$$

Es wird deutlich, dass diese Kosten denen der konventionellen Berechnungsvariante entsprechen.

Für die zweite Phase findet keine Unterscheidung der Blöcke statt, da in jedem Fall k Teilmatrizen mit x^2 Elementen existieren, die von $k-1$ Knoten an einen Anderen gesendet werden müssen. Bei insgesamt k^2 Blöcken entstehen dadurch Kosten in Höhe von $x^2k^2 \cdot (k-1)$.

Die Gesamtkosten dieser Berechnungsart sind zusammenfassend ermittelbar über

$$n^2k - xn + x^2k^2 \cdot (k-1).$$

Das Umformen der Formel führt zu den gleichen Kosten wie bei der konventionellen Art:

$$n^2k - xn + n^2k - n^2.$$

Kosten für andere Operationen

Wie einführend beschrieben, sollen die Partitionierungsstrategien möglichst robust gegenüber anderen Operationen sein. Zur Überprüfung dessen werden in diesem Abschnitt die Kosten der vorgestellten Strategien für $B \cdot A$ (1) und $A^T \cdot B$ (2) betrachtet.

Betrachtung für (1) Für die drei Strategien zeilenweise, spaltenweise und blockweise ist intuitiv zu erkennen, dass keine Änderung der Partitionierung auftritt. Folglich sind die Kosten für $A \cdot B$ und $B \cdot A$ gleich. Lediglich bei der zeilenweise-spaltenweise Variante ergibt sich eine andere Verteilung, deren Kosten hier berechnet werden.

Aus der für $A \cdot B$ geltenden zeilenweise-spaltenweise Partitionierung wird beim Vertauschen der Matrizen eine spaltenweise-zeilenweise Partitionierung. Die zugehörige Partitionierung der 4×4 Beispielmatrizen ist durch die am Anfang des Abschnitt eingeführte Darstellung repräsentiert.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{pmatrix} =$$

$$\begin{pmatrix} (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) \\ (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) \\ (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) \\ (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) & (1,1),(2,2),(3,3),(4,4) \end{pmatrix}$$

Es ist zu erkennen, dass in der Multiplikationsphase kein Senden erforderlich ist, da die benötigten Faktoren immer auf gleichen Knoten liegen. Nach den dort ausgeführten Multiplikationen existieren n Ergebnisse, wovon $m - 1$ zu einem anderen Knoten geschickt werden müssen. Die Gesamtkommunikationskosten sind demnach $n^2 \cdot (m - 1)$.

Betrachtung für (2) Im Fall der Transponierung von A entstehen für die Strategien zeilenweise, spaltenweise und zeilen-spaltenweise bereits bekannte Varianten: spalten-zeilenweise, zeilen-spaltenweise und ebenfalls spaltenweise Partitionierung. Die Kosten für diese Fälle wurden in den vorherigen Abschnitten bereits ermittelt.

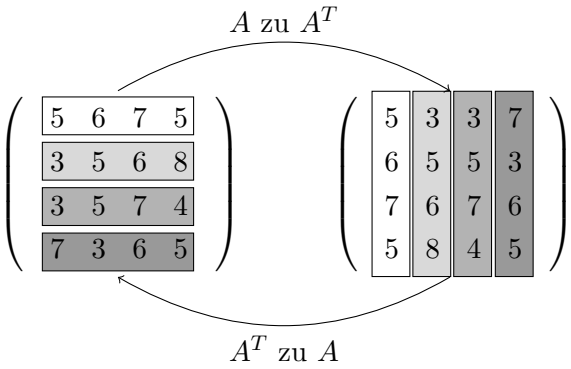

Abbildung 5.10: Transponierung von A

Bei der Transponierung einer Matrix mit Blockpartitionierung ergeben sich jedoch Unterschiede zu den bereits ermittelten Kosten von $A \cdot B$, weshalb hier die Berechnung für $A^T \cdot B$ anhand der Blockmatrixrechnung vorgenommen wird. Nach der Transponierung von A ergibt sich folgendes Schema:

$$\begin{bmatrix} A_1^T & A_3^T \\ A_2^T & A_4^T \end{bmatrix} \cdot \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} =$$

$$\begin{bmatrix} A_1B_1 + A_2B_2 & A_1B_3 + A_2B_4 \\ A_3B_1 + A_4B_2 & A_3B_3 + A_4B_4 \end{bmatrix}$$

Wie bei der Berechnung von $A \cdot B$ ist hier zwischen Diagonal- und anderen Blöcken zu unterscheiden. Für die Diagonalblöcke sind hier im Gegensatz zu $A \cdot B$ sogar alle Faktorenpaare auf gleichen Knoten, weshalb in der Multiplikationsphase für diese k Blöcke keine Kommunikation nötig ist. Für jede Blockspalte, die an alle Zeilenknoten gesendet wird, tritt dies genau einmal auf, weshalb von Formel 5.3 k^2 abgezogen werden muss. Nach der Multiplikation mit x^2 Elementen pro Block ergeben sich $x^2k^3 - x^2k^2 = n^2k - n^2$ zu versendende Elemente in der Multiplikationsphase. Die Kosten in der Summationsphase sind unverändert zu $A \cdot B$ mit $n^2k - n^2$ berechenbar.

Da $x = n/k$ ist, kann xn maximal gleich groß wie n^2 sein. Dies bedeutet, dass die transponierte Variante sogar besser ist, als die für $A \cdot B$.

Zusammenfassung der Kostenberechnungen

In diesem Abschnitt werden die Ergebnisse der vorherigen Abschnitte zusammengefasst und anschließend auf Grundlage dieser ein Fazit gezogen. Als erste Gegenüberstellung dient die nachfolgende Tabelle 5.1, welche die Kosten für $A \cdot B$ auflistet.

Strategie	**Multiplikationsphase**	**Summationsphase**	**Gesamtkosten**
Zeilen	$n^2m - n^2$	-	$n^2m - n^2$
Spalten	$n^2m - n^2$	-	$n^2m - n^2$
Zeilen-Spalten	$n^2m - n^2$		$n^2m - n^2$
Spalten-Zeilen	-	$n^2m - n^2$	$n^2m - n^2$
Block (konventionell)	$n^2k - xn$	$n^2k - n^2$	$n^2k-xn+n^2k-n^2$
Block (Blockmatrixrechnung)	$n^2k - xn$	$n^2k - n^2$	$n^2k-xn+n^2k-n^2$

Tabelle 5.1: Kosten der Strategien für $A \cdot B$

Aus dieser Übersicht wird deutlich, dass sich die Kommunikationskosten innerhalb der Zeilen- und Spaltenpartitionierungen nicht unterscheiden. Lediglich die Blockpartitionierung führt zu einer Verbesserung der Kommunikationskosten.

Die Robustheit der einzelnen Partitionen wurde mittels der Kommutation von A und B, sowie der Transponierung von A untersucht. Die folgende Tabelle zeigt dabei die ermittelten Formeln für das Vertauschen von A und B.

Strategie	**Multiplikationsphase**	**Summationsphase**	**Gesamtkosten**
Zeilen	unverändert zu 5.1		
Spalten	unverändert zu 5.1		
Zeilen-Spalten	entspricht Spalten-Zeilen aus 5.1		
Spalten-Zeilen	entspricht Zeilen-Spalten aus 5.1		
Block	unverändert zu 5.1		

Tabelle 5.2: Kosten der Strategien für $B \cdot A$

Aus der Tabelle ist die Robustheit aller vorgestellten Partitionierungsstrategien gegenüber der Kommutation ersichtlich, da in keinem Fall andere Kommunikationskosten auftreten. Bei der Transponierung ergeben sich hingegen Unterschiede, wie die nachstehende Auflistung verdeutlicht.

Strategie	**Multiplikations-phase**	**Summations-phase**	**Gesamtkosten**
Zeilen	entspricht Spalten-Zeilen aus 5.1		
Spalten	entspricht Zeilen-Spalten aus 5.1		
Zeilen-Spalten	entspricht Spaltenpartitionierung		
Spalten-Zeilen	entspricht Zeilenpartitionierung		
Block	$n^2k - n^2$	$n^2k - n^2$	$n^2k - n^2 + n^2k - n^2$

Tabelle 5.3: Kosten der Strategien für $A^T \cdot B$

Hier ergibt sich bei der Blockpartitionierung eine Veränderung in den Kosten - allerdings zum Positiven. Nach der Transponierung von A sind alle Diagonalblöcke lokal berechenbar, wodurch sich die Anzahl der zu versendenden Elemente verringert.

In der Abbildung 5.11 sind die Kosten der Spalten-, Zeilen-, Spalten-Zeilen- und Zeilen-Spaltenpartitionierung denen der Blockpartitionierung für eine Konfiguration mit 9 Knoten gegenüber gestellt.

Durch die Linienverläufe wird der geringere Sendeaufwand bei der Verwendung der Blockpartitionierung gegenüber den anderen Varianten deutlich. Darüber hinaus ist die Verbesserung durch Transponierung im Fall der Blockverteilung ersichtlich.

Die Analyse der verschiedenen Partitionierungsarten hinsichtlich der Anzahl zu versendender Elemente führt zu mehreren Ergebnissen. Die Partitionierungsarten spaltenweise, zeilenweise, spalten-zeilenweise und zeilen-spaltenweise haben alle die gleiche Kostenformel. Eine Veränderung der Matrixreihenfolge beziehungsweise die Transponierung einer Matrix führt zu einer dieser vier Partitionierungsarten, weshalb dabei keine Veränderung der Kosten entsteht. Die Anzahl der zu versendenden Elemente ist bei der Blockpartitionierung sowohl mit der konventionellen, als auch der Blockmatrixrechnung gleich und dabei geringer, als die der anderen Partitionierungsarten. Bei der Kommutation entsteht keine Veränderung der Kostenformel, dafür jedoch bei der Transponierung. Diese führt allerdings zu einer Verbesserung dieser. Es lässt sich daher feststellen, dass alle Varianten hinsichtlich der Kosten robust sind.

Die Berechnungen beruhen auf der Annahme, dass nur die nötigsten Elemente gesendet und die lokal möglichen Operationen dort auch ausgeführt werden. Somit zeigen diese Kostenformeln an, bei welcher Partitionierungsart der Optimierer das meiste Potenzial hat, um Kommunikation zu vermeiden. Im nachfolgenden Abschnitt soll daher unter Anderem geprüft werden, ob dieses von Postgres-XL ausgeschöpft wird.

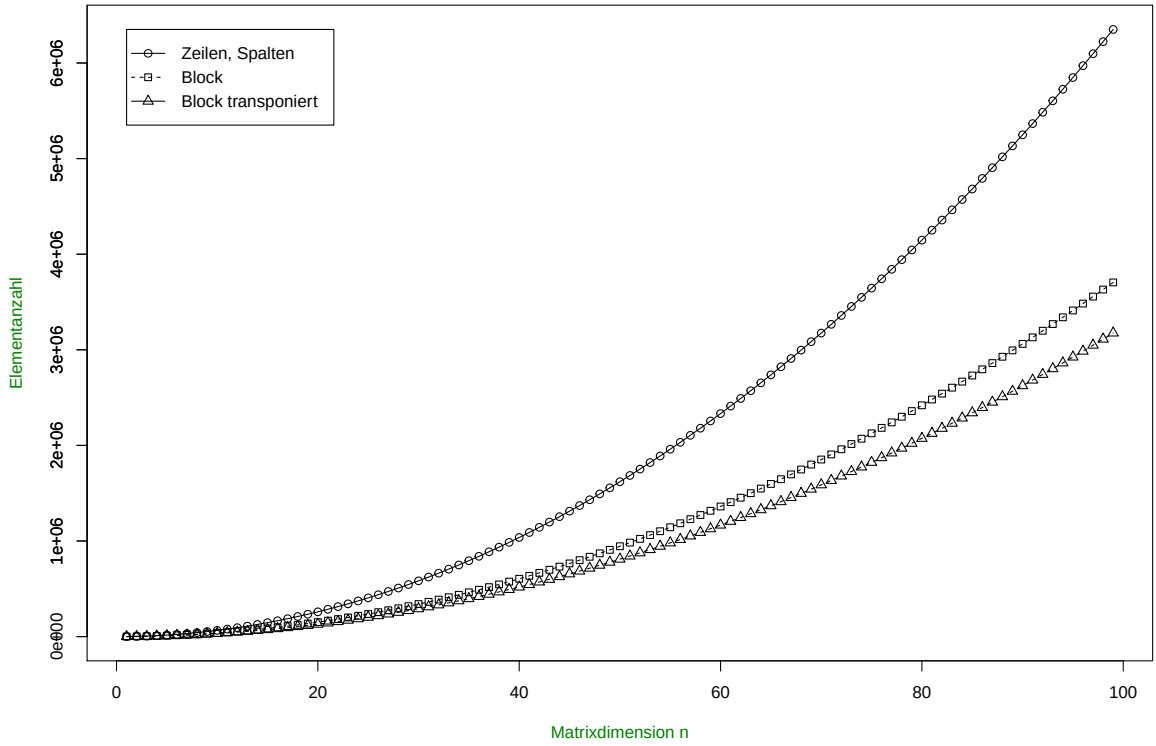

Abbildung 5.11: Elementanzahl der Partitionierungsstrategien im Vergleich

5.5 Implementation mit Postgres-XL

Auf Basis der in Abschnitt 5.4.1 vorgestellten und in Bereich 5.4.2 analysierten Partitionierungsstrategien soll die Matrixmultiplikation exemplarisch, mit dem in Abschnitt 4.7 vorgestellten parallelen Datenbanksystem Postgres-XL, umgesetzt werden.

Aus den vorherigen Erkenntnissen lassen sich mehrere Kernfragen ableiten, die in diesem Abschnitt Untersuchung finden. Aus dem *Stand der Technik* wird deutlich, dass Interanfrage- und Interoperatorparallelität bereits Bestandteil herkömmlicher paralleler Datenbanksysteme sind. Mit der Möglichkeit zur Partitionierung einzelner Relationen in Postgres-XL ist jedoch zusätzlich die Grundlage für Intraoperatorparallelität im Datenbankbereich geschaffen worden. Hier ist demnach zu untersuchen, ob diese Option vom Datenbankoptimierer bereits bei den Anfragen aus 5.2 sowie 5.3 erkannt und im Anfrageplan umgesetzt wird. Darüber hinaus ist zu untersuchen, inwiefern die betrachteten Partitionierungsstrategien den Anfrageplan und die Bearbeitungszeit beeinflussen.

Zur Klärung dieser Kernfragen wird mit Postgres-XL die Multiplikation auf zwei zeilenweise und blockweise partitionierten Matrizen mit der in 5.2 gezeigten Anfrage für verschiedene Dimensionen umgesetzt und analysiert. Ebenfalls wird als komplexere Operation der Vorwärtsalgorithmus (siehe Abschnitte 3.3.1 und 5.3) mit einer Blockpartitionierung unter den gleichen Gesichtspunkten betrachtet.

Dieser Abschnitt strukturiert sich in sechs Unterpunkte. Zunächst werden die Details des Postgres-XL-Datenbankclusters in 5.5.1 dargestellt, bevor eine Erläuterung zur Umsetzung der Partitionierungsstrategien stattfindet. Dem folgt die Betrachtung der Anfragepläne und Darstellung der Laufzeiten in den Abschnitten 5.5.3 und 5.5.4. Abschließend werden in 5.5.5 und 5.5.6 die Ergebnisse validiert und ein Fazit auf Grundlage der Erkenntnisse gezogen.

5.5.1 Hardwareübersicht

Das im Rahmen der Rostock Massive Data Research Facility zur Verfügung stehende Postgres-XL Cluster besteht aus insgesamt 9 Knoten, wobei die installierten Versionen Postgres-XL 9.5r1.4 (Cluster) und PostgreSQL 9.5.5 (Knoten) sind. Von der standardmäßigen Installation weichen dabei die in Tabelle 5.4 aufgelisteten Punkte ab.

Effective Cache Size	4 GB
Worker Memory	512 MB
Maintenance Memory	1 GB
Temporary Buffer	64 MB
Shared Buffer	1 GB
Segment Size	1 GB

Tabelle 5.4: Geänderte Hardwareparameter

Die Netzgeschwindigkeit im Cluster beträgt 10 Gbit/s. Da auf den Knoten PostgreSQL in der Version 9.5.5 läuft, ist die in 4.6.1 vorgestellte Multicoreprozessorausnutzung nicht möglich.

5.5.2 Verteilung der Relationen

Relationen können in Postgres-XL auf vier verschiedene Arten partitioniert werden. Dabei basieren drei auf dem systemeigenen Befehl `DISTRIBUTE BY` [Thea] und die vierte Variante auf der bereits in PostgreSQL bekannten Vererbung von Tabellenstrukturen [Pos].

Über die Anweisung `DISTRIBUTE BY` ist es möglich, die Daten Round-Robin, per Hash und Modulo zu verteilen. Dabei ist es jedoch nicht möglich, die Hashfunktion selbst anzulegen.

Stattdessen wird eine interne Funktion verwendet, die anhand der Werte einer gewählten Spalte die Tupel auf entsprechende Knoten verteilt, was der aus 4.3.1 bekannten Bereichspartitionierung ähnelt. Beim Modulomodus werden die Werte der gewählten Spalte mit der Knotenanzahl Modulo gerechnet. Im Round-Robin-Verfahren erfolgt die Zuordnung der einzelnen Tupel zum Knoten hingegen über die Reihenfolge ihres Hinzufügens.

Die letzte Strategie basiert auf der Vererbung von Relationen, die so auch in PostgreSQL existiert, nur dass hier zusätzlich die Kindtabellen auf verschiedene Knoten gelegt werden können. Diese Art benötigt eine Mastertabelle von der die Kindtabellen ihre Struktur erben und an die, für gewöhnlich, Anfragen gestellt werden. Dabei ist die Mastertabelle meist leer und die Daten liegen in den Kindtabellen. Regeln ermöglichen es über Attributwerte, auch für mehrere Attribute, zu steuern, in welche Kindtabelle die Daten eingefügt werden. Folglich sind mit dieser Variante alle vorgestellten Partitionierungsstrategien umsetzbar. Dennoch wird im Folgenden nur die Blockpartitionierung mit dieser Form umgesetzt, da die Zeilenverteilung über `DISTRIBUTE BY MODULO(row)` in Postgres-XL intern besser verwertbar scheint. Um mögliche Vorteile durch die vorimplementierte Funktion `DISTRIBUTE BY` auch bei der Blockpartitionierung auszunutzen, werden die Daten in gewünschter Reihenfolge im Round-Robin-Verfahren in die Relation eingefügt.

Matrixmultiplikation

Zur Analyse der Matrizenmultiplikation werden die Relationen zeilenweise via Modulo und blockweise über Round-Robin, sowie mittels der Vererbungsmethode realisiert. In allen Fällen liegen auf jedem Knoten 1/9 der Gesamtelemente.

Zeilenweise mit Modulo

Eine Relation, bei der die Daten zeilenweise verteilt werden, lässt sich mit dem folgenden Befehl erzeugen.

```
CREATE TABLE zeilenweise (row INTEGER, col INTEGER, val DOUBLE
   PRECISION) DISTRIBUTE BY MODULO(row)
```

Beim Einfügen neuer Daten entscheidet das Ergebnis von `MODULO(Attributwert von row)` über den Speicherort des Tupels.

Blockweise mit Vererbung

Das Erstellen der Blockpartitionierung über die Vererbung von Relationen erfolgt in zwei Schritten. Im ersten Schritt müssen die einzelnen Relationen erstellt werden, was hier pro

Matrix eine Mastertabelle und neun Kindtabellen sind (entsprechend der Knotenanzahl). Diese lassen sich, hier beispielhaft für $n = 900$ und eine Kindtabelle, über folgende Befehle erzeugen.

```
CREATE TABLE blockMaster (row INTEGER, col INTEGER, val DOUBLE
   PRECISION);
CREATE TABLE blockChild1 (CHECK row>0 AND row<=300 AND col>0
   AND col<=300) INHERITS(blockMaster) TO NODE(dn_1);
```

Im zweiten Schritt müssen Regeln erstellt werden, welche die Daten beim Einfügen an die korrekte Kindtabelle delegieren.

```
CREATE OR REPLACE RULE AS ON INSERT TO blockMaster WHERE row>0
   AND row<=300 AND col>0 AND col<=300 DO INSTEAD INSERT INTO
   blockChild1 VALUES (NEW.*);
```

Blockweise mit Round-Robin

Analog zu Modulovariante kann mit folgendem Befehl eine Relation erstellt werden, bei der die Daten nach der Reihenfolge des Hinzufügens verteilt werden.

```
CREATE TABLE blockweiseRoundRobin (row INTEGER, col INTEGER,
   val DOUBLE PRECISION) DISTRIBUTE BY ROUNDROBIN
```

Um über diese Methode eine Blockpartitionierung zu erreichen, werden die Daten, beispielsweise für eine Matrix mit $n = 900$, nach folgendem Schema eingefügt:

```
(1,1,a_{1,1}), (1,301,a_{1,301}), (1,601,a_{1,601}), (301,1,a_
   {301,1}), (301,301,a_{301,301}), (301,601,a_{301,601}),
   (601,1,a_{601,1}), (601,301,a_{601,301}), (601,601,a_
   {601,601}) ...
```

Es ist erkennbar, dass nur für die Modulovariante die Struktur der Importdatei keinen Einfluss auf die Verteilung hat und Daten für jede Dimension ohne weitere Bearbeitung einfügbar sind. Beim Vererbungsfall hingegen muss für jede Matrixgröße eine Anpassung der Einfügeregeln erfolgen und im Round-Robin-Modus die Importdatei verändert werden.

Vorwärtsalgorithmus

Die Berechnung der Wahrscheinlichkeit einer Beobachtungssequenz in einem HMM (siehe 3.3) mit dem Vorwärtsalgorithmus benötigt die Übergangswahrscheinlichkeitsmatrix A, Beobachtungswahrscheinlichkeitsmatrix B, den Anfangsverteilungsvektor π und Observationsvektor `obs`.

Die beiden Matrizen werden blockweise über die Round-Robin-Variante (siehe 5.5.2) realisiert, wohingegen die Verteilung beider Vektoren, welche im Rahmen dieser Arbeit einspaltige Matrizen sind, normal mittels Round-Robin erfolgt.

5.5.3 Analyse der Anfragepläne

Die Analyse der Anfragepläne soll klären, ob der Optimierer das Potenzial, einen Operator zu parallelisieren, erkennt und folglich einen Plan erstellt, der Teilanfragen an Knoten weiterleitet, auf deren Ergebnissen anschließend die Bearbeitung fortgesetzt wird.

Matrixmultiplikation

Dabei ergeben sich bei der Matrixmultiplikation zwischen den Plänen mit `DISTRIBUTE BY`-Relationen und Vererbungsrelationen Unterschiede. Die Pläne der `DISTRIBUTE BY`-Tabellen sind hingegen, unabhängig von Round-Robin oder Modulo, gleich.

Vererbung

Aus Gründen der Übersichtlichkeit ist der entscheidende Anteil des Plans in Abbildung 5.12 verdeutlicht, wobei der gesamte Plan im Anhang in Programmbeispiel A.1 einzusehen ist. Die Abbildung und der Plan basieren auf Matrixrelationen der Dimension $n = 900$ und 810000 Tupeln.

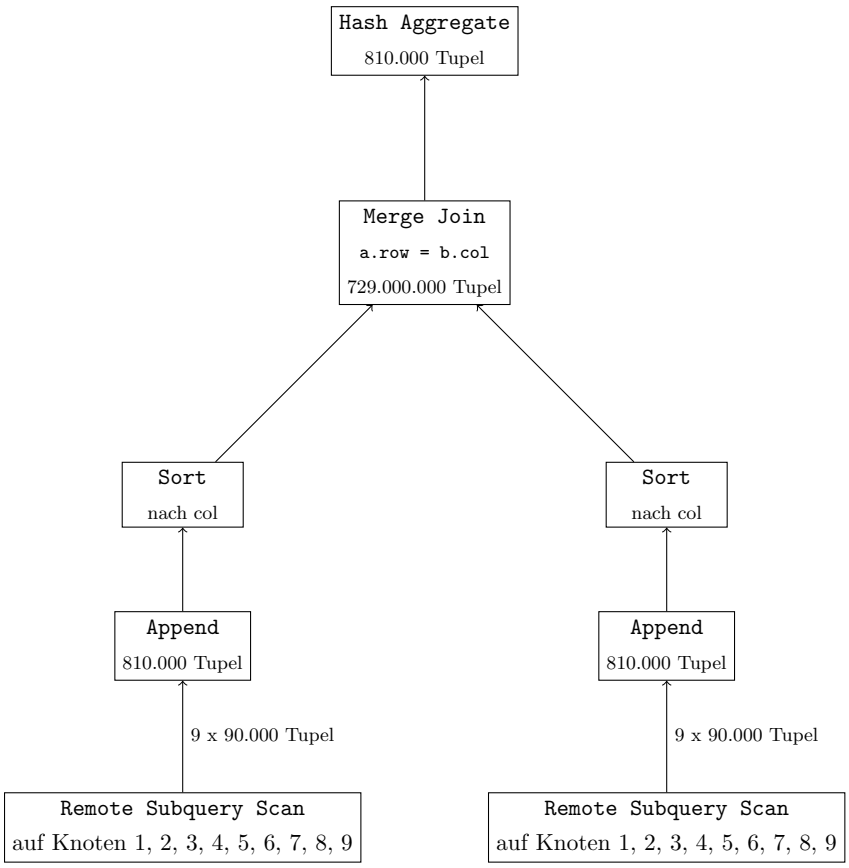

Abbildung 5.12: Anfrageplan Matrixmultiplikation bei der Vererbungsvariante

Aus dem Plan ist ableitbar, dass alle Daten von den Datenknoten geholt und an einem Ort zusammengefasst werden. An diesem erfolgt dann der Verbund zwischen den Relationen und die anschließende Aggregation. Intraoperatorparallelität ist aus den vorhandenen Informationen nicht ableitbar.

Modulo und Round-Robin

Betrachtet wird hier der Plan mit Matrizen der Dimension $n = 900$. Dabei ergeben sich bei diesen Varianten keine Unterschiede im erstellten Abarbeitungsplan, welcher im nachstehenden Programmbeispiel 5.11 dargestellt ist. Um die Übersichtlichkeit zu wahren, wurden die Kostenschätzungen entfernt und nur die aktuellen Kosten aufgeführt.

```
1 HashAggregate (actual time=72180.348..72432.103 rows=810000 loops=1)
2   Output: a."row", b.col, sum((sum((a.val * b.val))))
3   Group Key: a."row", b.col
4   ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn
        _9) (actual time=58400.427..66192.068 rows=7290000 loops=1)
5     Output: a."row", b.col, sum((a.val * b.val))
```

Programmbeispiel 5.11: Plan für Modulo und Round-Robin verteilte Matrizen

Der Plan setzt sich aus dem inneren `Remote Subquery Scan` und einem äußeren `Hash Aggregate` zusammen. In beiden Fällen besteht die Ausgabe aus den Attributen `a.row`, `b.col` und dem Aggregat `sum(a.val *b.val)`. Aus der Darstellung `sum((sum((a.val *b.val))))`, der äußeren Abarbeitung, lässt sich ableiten, dass hier bereits teilweise aggregierte Summen verarbeitet werden. Die Anzahl der Tupel, die das `Remote Subquery` weitergibt, entspricht mit 7290000 der Knotenanzahl multipliziert mit der Relationskardinalität einer Matrix.

Bei der Abfrage des Plans mit `EXPLAIN ANALYZE` wird diese tatsächlich ausgeführt und der verwendete Plan angezeigt. Jedoch werden in diesem nicht die mit `Remote Subquery Scan` bezeichneten Unteranfragen, die auf den Knoten stattfinden, ausgegeben. Diese sind allerdings mit `EXPLAIN` ersichtlich, wobei hier nicht sichergestellt ist, dass dieser Plan bei der Abfrage benutzt wird. In 5.12 ist der mit `EXPLAIN` erzeugte Subplan gezeigt.

```
1  ->  HashAggregate
2    Group Key: a."row", b.col
3    ->  Hash Join
4    Hash Cond: (b."row" = a.col)
5    ->  Remote Subquery Scan on all (dn_1,dn_2,...,dn_9)
6      Distribute results by H: "row"
7      ->  Seq Scan on brr b
8    ->  Hash
9      ->  Remote Subquery Scan on all (dn_1,dn_2,...,dn_9)
10       Distribute results by H: col
11       ->  Seq Scan on arr a
```

Programmbeispiel 5.12: Subplan von Modulo und Round-Robin

Auch dieser Plan besteht wiederum aus zwei Ebenen. Auf der zweiten, inneren Ebene erfolgt ein `Hash Join`, dessen Ergebnis eine Schicht weiter oben über `HashAggregate` aggregiert wird. Von Bedeutung ist hier vor allem die sechste und zehnte Zeile, da dort eine Repartitionierung der Relationen nach ihrem `JOIN`-Attribut stattfindet. Folglich ist ab diesem Schritt die gewählte Block- oder Zeilenverteilung nicht mehr vorhanden und beide Varianten agieren gleich. Ein Unterschied in der Laufzeit (siehe Abschnitt 5.5.4) dürfte demnach nur aus dem Datenbeschaffungs- und Repartitionierungsschritt resultieren.

Die Betrachtung der beiden Anfragepläne verdeutlicht, dass mit einem parallelen Datenbanksystem, hier Postgres-XL, Matrixmultiplikation in einem gewissen Maße mit Intraoperatorparallelisierung abgearbeitet wird. Dabei scheint der Optimierer jedoch nur bei der Benutzung des eigenen `DISTRIBUTE BY`-Befehls die Verteilung auszunutzen. Der Repartitionierungsschritt

zeigt darüber hinaus eine fehlende Differenzierung zwischen den gewählten Partitionierungsstrategien, sowie die Nichtausnutzung von bereits lokal möglichen Operationen.

Vorwärtsalgorithmus

Der in Abbildung 5.13 visualisierte Anfrageplan (als Programmbeispiel in A.2) basiert auf der geschachtelten Anfrage 5.3 mit drei Beobachtungen. Weiterhin haben beide Matrizen die Dimension $n \times n$ mit $n = 900$, weshalb der Anfangsverteilungsvektor die Länge 900 hat. Zum besseren Verständnis sind hier Abschnitte, welche einen Anfrageteil darstellen, farblich gekennzeichnet und in Abbildung 5.14 zugeordnet.

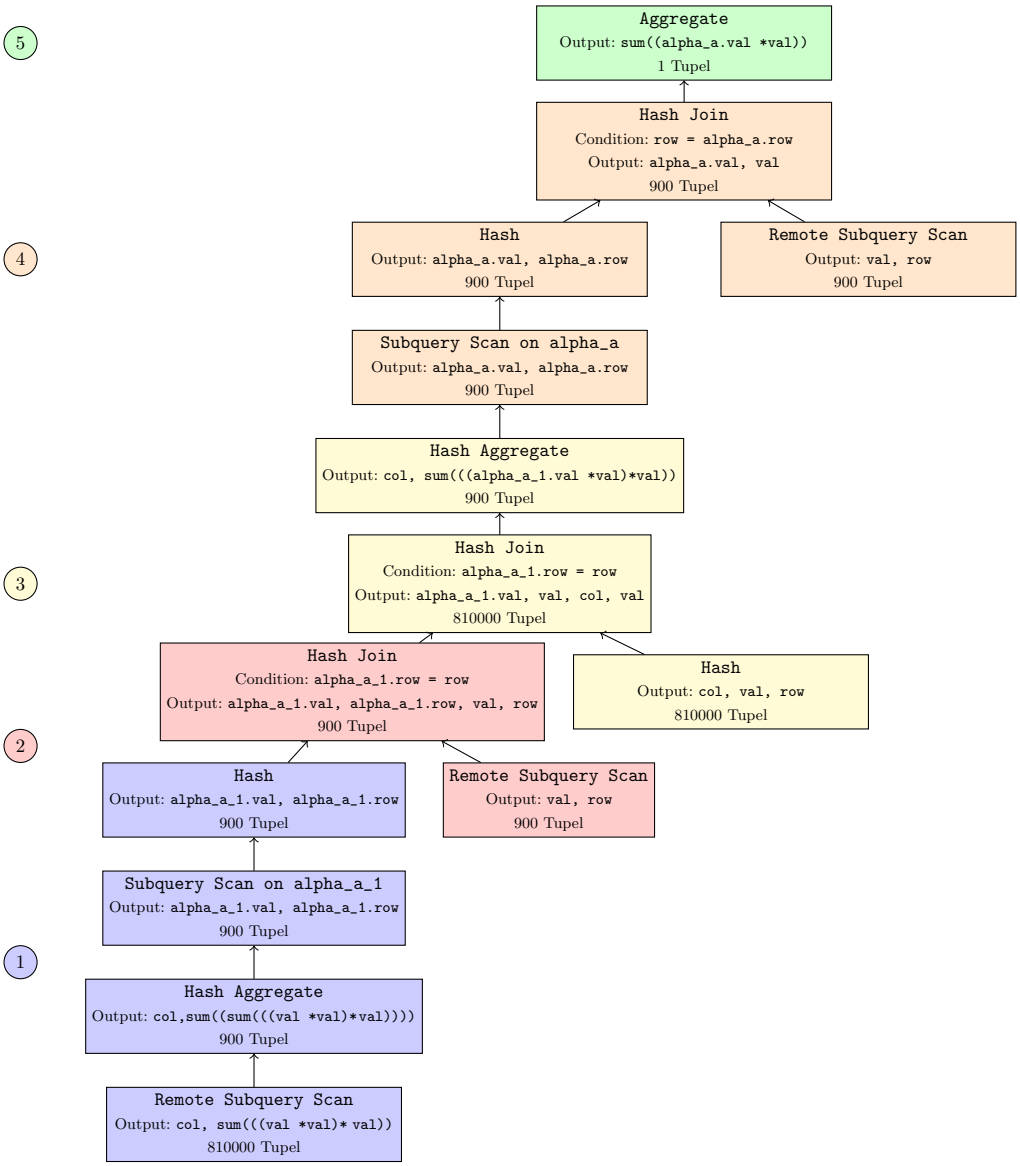

Abbildung 5.13: Anfrageplan Vorwärtsalgorithmus

1

```
(SELECT A.col as row, SUM(aa.val * A.val) as val FROM
(SELECT Pi.row as row, Pi.val * B.val as val FROM intra.pi900 as PI
    JOIN intra.hmmb900 as B on Pi.row = B.row WHERE B.col = 1) as aa
JOIN hmmarr as A ON aa.row = A.row GROUP BY A.col) alpha_a
```

2

```
(SELECT alpha_a.row as row, alpha_a.val * B.val AS val FROM
(QUERY 1)
alpha_a JOIN intra.hmmb900 as B on alpha_a.row = B.row WHERE B.col =
    2)
```

3

```
(SELECT A.col as row, SUM(aa.val * A.val) as val FROM
(QUERY 2)
JOIN hmmarr as A ON aa.row = A.row GROUP BY A.col) alpha_a
```

4

```
(SELECT alpha_a.row as row, alpha_a.val * B.val as val FROM
(QUERY 3)
JOIN intra.hmmb900 as B on alpha_a.row = B.row WHERE b.col = 3)
```

5

```
SELECT SUM(val) FROM
(QUERY 4)
as complete_query;
```

Abbildung 5.14: Zuordnung von Baum- zu Anfrageabschnitt

Aus den einzelnen Bestandteilen des Anfrageplans lässt sich eine sequentielle Abarbeitungsmechanik feststellen. Einzelne Operatoren, wie zum Beispiel die Summation im `Remote Subquery Scan` der Anfrage 1, werden zwar auf Partitionen geschoben, jedoch findet keine Aufteilung der Gesamtanfrage auf Teilrelationen statt. Dies bedeutet, dass zwar für kleinere Bestandteile eine Intraoperatorparallelisierung stattfindet, allerdings nicht für den Algorithmus im Gesamten.

5.5.4 Laufzeit der Anfragen

In diesem Abschnitt findet die Laufzeitanalyse der Matrizenmultiplikation und des Vorwärtsalgorithmus statt. Dabei werden für die Matrizenmultiplikation Abarbeitungszeiten für unterschiedliche Dimensionen und die verschiedenen Partitionen durchgeführt. Beim Vorwärtsalgorithmus werden hingegen die Länge der Beobachtungssequenz und Matrixdimension variiert.

Matrixmultiplikation

Die Abbildung 5.15 visualisiert vergleichend die Laufzeiten der Multiplikation zweier Matrizen mit den Dimensionen 900, 1200, 1500, 1800 und 3000 für die vorgestellten Partitionierungsstrategien. Hierbei sind die Werte des Attributs `val` zufällige `DOUBLE PRECISION` Zahlen zwischen 1 und 1000.

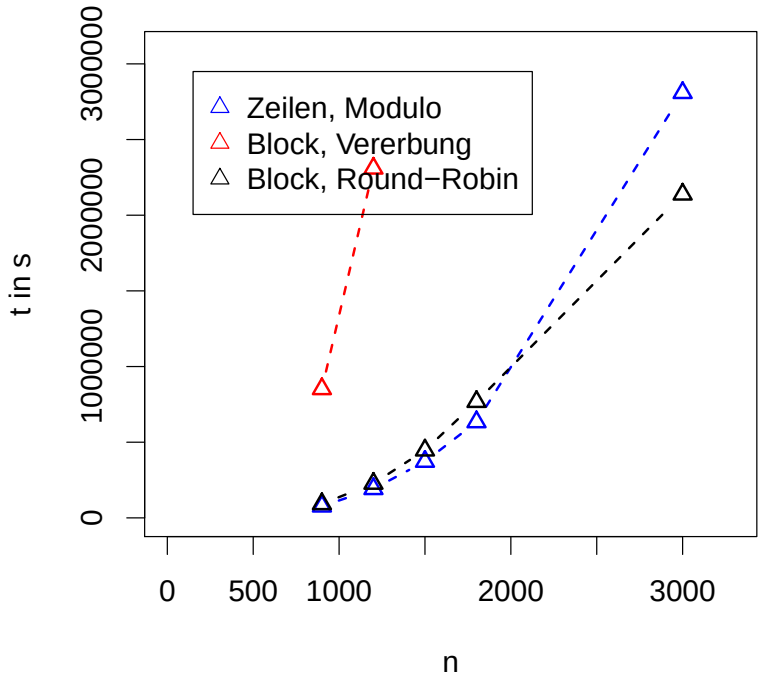

Abbildung 5.15: Laufzeit der Matrixmultiplikation bei verschiedenen Partitionierungen

Die Varianten Blockweise über Round-Robin und Zeilenweise über Modulo haben bei jeder Dimension ähnliche Laufzeiten, wobei bis zu $n = 1800$ Zeilenweise geringfügig schneller als Blockweise ist. Mögliche Gründe lassen sich jedoch nicht aus den Anfrageplänen (siehe 5.5.3) erkennen.

Die Laufzeiten der Vererbungsmethode sind, wie ausgehend vom Anfrageplan zu erwarten, wesentlich langsamer als die beiden anderen Versionen. Ab der Dimension $n = 1500$ musste die Analyse nach sieben Stunden ohne Ergebnis abgebrochen werden.

Vorwärtsalgorithmus

Die Laufzeiten für den Vorwärtsalgorithmus (siehe Abbildung 5.16) basieren auf Transitions- und Beobachtungswahrscheinlichkeitsmatrizen der Dimension 900, 1200, 1500, 1800 und 3000. Dabei werden für jede Matrixdimension Beobachtungssequenzen der Länge 5, 6, 7, 8, 9 und 10 betrachtet.

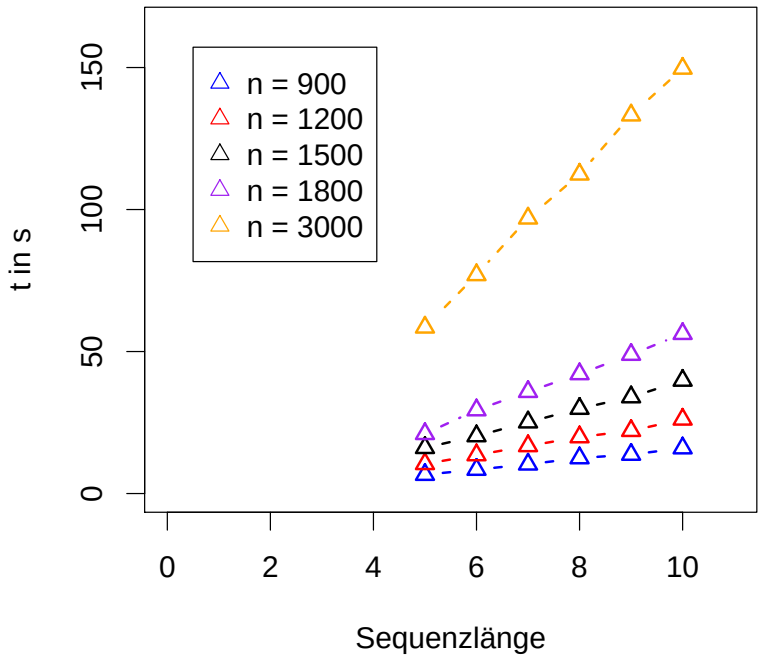

Abbildung 5.16: Laufzeit des Vorwärtsalgorithmus bei verschiedenen Beobachtungssequenzlängen

Anhand der grafischen Darstellung ist bei allen Dimensionen eine lineare Gerade bezüglich der Observationslänge zu erkennen. Dabei erhöht sich der Anstieg dieser je Dimension, da die zu bearbeitende Datenmenge steigt. Weiterhin lässt sich daraus für diesen Dimensions- und Observationsbereich ableiten, dass keine Anpassung des Anfrageplans aufgrund größerer Elementanzahlen stattfindet.

5.5.5 Validität der Ergebnisse

Zur Überprüfung, ob die erzeugten Ergebnisse korrekt sind, wird hier eine zweiteilige Validierung vorgenommen. Dabei gilt es zunächst sicherzustellen, dass die Relationen wie gewünscht auf die vorhanden Knoten verteilt sind. Anschließend erfolgt die Kontrolle der berechneten Ergebnisse des hier vorgestellten Vorwärtsalgorithmus und der Matrixmultiplikation.

Partitionierung

Der erste Fall lässt sich über SQL-Anfragen klären, die nur auf einzelnen Knoten ausgeführt werden und nur die dort vorhanden Daten anzeigen. Zusammen mit `WHERE`-Klauseln, welche auf die gewünschten Blöcke beziehungsweise Zeilen einschränken, lässt sich die Gewünschte mit der tatsächlichen Anzahl an Elementen vergleichen.

Stellvertretend findet die Darstellung zweier Validierungsanfragen, jeweils eine für Block- sowie Zeilenverteilung, auf der Matrixdimension $n = 900$, statt. Für den Block A_4 gelten beispielsweise die Bedingungen $row > 300$, $row <= 600$, $col <= 300$ und die Knotennummer 4, welche mit der nachfolgenden SQL-Anfrage überprüft werden:

```
1 EXECUTE DIRECT ON (dn_4) 'SELECT COUNT(*) FROM arr900 WHERE row
    > 300 AND row <= 600 AND col <= 300';
```

Diese Abfrage ermittelt `COUNT = 90000`, was im Zusammenhang mit der Matrixdimension von 900 eine korrekte Partition A_4 bedeutet.

Im Fall der Zeilenverteilung ist die `WHERE`-Klausel anzupassen, da hier die einzelnen Zeilen, welche auf dem Knoten vorhanden sein sollen, Betrachtung finden. Korrekt ist die Partitionierung für einen Knoten, wenn die Abfrage 90000 Tupel errechnet.

```
1 EXECUTE DIRECT ON (dn_4) 'SELECT COUNT(*) FROM am900 WHERE row %
     9 = 3';
```

Diese Anfragen wurden für die verschiedenen Dimensionen und Partitionierungsarten ausgeführt, wodurch die Richtigkeit der Verteilung hier verwendeter Relationen bestätigbar ist.

Berechnung

Die Validierung der berechneten Matrixmuliplikationsergebnisse erfolgt in zwei Schritten. Als erstes wird mit dem Statistikprogramm *R* über den vorimplementierten Matrixoperator [R C16] eine Ergebnismatrix erzeugt, um diese anschließend in Postgres-XL zu importieren. Dort wird, im zweiten Schritt, über die SQL-Anfrage in 5.13 ermittelt, ob die beiden Ergebnismatrizen `c1` und `c2` übereinstimmen. Da die Attributwerte vom Datentyp `DOUBLE PRECISION` sind, wird eine Übereinstimmung angenommen, wenn das Ergebnis der Abfrage nah bei oder genau null ist.

```
1 SELECT SUM(c1.val * c1.val - c2.val * c2.val) FROM c1 JOIN c2 ON
      c1.row = c2.row AND c1.col = c2.col
```

Programmbeispiel 5.13: Validierungsanfrage Matrixmultiplikation

Die Resultate der Abfrage 5.13 für die verschiedenen Dimensionen sowie Verteilungen sind in nachfolgender Tabelle aufgeführt.

Dimension	Blockweise, Round-Robin	Blockweise, Vererbung	Zeilenweise, Modulo
900	0	0	0
1200	0	0	0
1500	0	-	0
1800	0	-	0
3000	0*	-	0*

Die dargestellten Ergebnisse bestätigen die Richtigkeit der Matrixmultiplikation für alle Partitionierungsarten und Dimensionen. Bei den mit einem Stern (*) markierten Varianten wurden die Werte von `c1.val` und `c2.val` manuell auf die vierte Kommastelle gerundet. In den vorherigen Varianten geschah dies, sowohl bei R als auch SQL automatisch.

Die Validierung der Ergebnisse des Vorwärtsalgorithmus erfolgte hingegen in R. Als Berechnungsvorschrift wurde der in [MH17] dargestellte Programmcode verwendet.

5.5.6 Fazit

Die Analyse der Anfragepläne und Laufzeiten sollte klären, inwieweit aktuelle Systeme, im speziellen Postgres-XL, die Möglichkeiten zur Intraoperatorparallelisierung erkennen und ausnutzen.

Dabei wurde zum einen ersichtlich, dass simple Aggregatfunktionen, wie zum Beispiel die Summation, auf einzelne Knoten verschoben werden. Hierbei findet demnach eine Intraoperator-

parallelisierung für diese Aggregation statt. Wenn sich eine Operation nur aus solchen Aggregaten zusammensetzt, wie bei der Matrixmultiplikation, ist auch die Parallelisierung dieser im Gesamten möglich. Zum anderen wurde durch den Repartitionierungsschritt die fehlende Ausnutzung der vorhanden Partitionierungsstrategien ersichtlich. Die Möglichkeiten, lokale Berechnungen durchzuführen und nur bestimmte Elemente zu senden, wurde von Postgres-XL nicht umgesetzt. Gründe können hier, neben der fehlenden Implementierung, die hohe Netzgeschwindigkeit sein, durch die ein Repartitionierungsschritt kostengünstig ist. Bei der Analyse des Vorwärtsalgorithmus wird der beschriebene Sachverhalt, dass die Operation nicht als Ganzes, sondern als Kombination von Einzeloperationen angesehen wird, deutlicher. Hier entspricht der Anfrageplan dem sequentiellen Abarbeiten der einzelnen Teilanfragen und Parallelisierung findet nur auf dieser Ebene statt. Mit der Verwendung der Vererbungsvariante zur Erstellung von Partitionen wurde der Schwachpunkt aufgedeckt, dass Postgres-XL Aggregatfunktionen nur bei Verwendung des eingebauten `DISTRIBUTE BY`-Befehls auf untere Ebenen verschiebt.

Die Laufzeitbetrachtung galt hier dem Vergleich zwischen den einzelnen Strategien und nicht der Gegenüberstellung mit anderen Systemen. Bei diesem zeigt sich bezüglich der Matrixmultiplikation ein deutlicher Unterschied zwischen der Vererbungsvariante und den Fällen mit `DISTRIBUTE BY`. Dies belegt die mögliche Verbesserung, die eine intraoperatorparallelisierte Anfrage gegenüber einem sequentiellen Plan bietet.

6 Zusammenfassung und Ausblick

Abschließend werden die Inhalte dieser Arbeit zusammengefasst dargestellt und bewertet. Auf Basis der gewonnenen Erkenntnisse erfolgt anschließend ein Ausblick. Das Ziel war es, die Intraoperatorparallelität frequenter Basisoperationen des wissenschaftlichen Rechnens in SQL zu untersuchen.

Hierfür fand der Einstieg mit der Identifizierung und Vorstellung typischer mathematischer Operationen im Bereich des wissenschaftlichen Rechnens statt. Daneben erfolgte die Erläuterung des aus dem Gebiet der Aktivitäts- und Intentionserkennung bekannten Vorwärtsalgorithmus. Weitere Grundlagen wurden mit der Relationenalgebra und der hier verwendeten Datenbankanfragesprache SQL präsentiert.

Diesen Grundlagen folgte die Darstellung bekannter Architekturen, Techniken und Probleme, welche im allgemeinen Kontext der Parallelisierung, aber auch im speziellen Bereich der Intraoperatorparallelität, vorhanden sind. Als spezielle Systeme, welche in dieser Arbeit Verwendung finden, wurden PostgreSQL und Postgres-XL vorgestellt.

An die Klärung der Grundlagen und Vorstellung von bisherigen Ansätzen dieses Forschungsgebietes schloss sich im ersten Umsetzungsabschnitt die Festlegung eines Matrixrelationenschemas und die darauf basierende Entwicklung von SQL-Anfragen für die identifizierten mathematischen Operationen und den Vorwärtsalgorithmus an. Dabei stellte sich heraus, dass die rekursive Variante des Vorwärtsalgorithmus mit PostgreSQL und Postgres-XL nicht realisierbar ist und in Postgres-XL Rekursion im Allgemeinen nicht auf verteilten Relationen möglich ist.

In der Folge wurden verschiedene Partitionierungsstrategien hinsichtlich der zu versendenden Elemente bei der Matrixmultiplikation betrachtet. Bei den untersuchten Partitionierungsstrategien zeigte sich, dass die Zeilen-, Spalten- und Zeilen-Spaltenpartitionierung die gleiche Anzahl an zu versendenden Elementen benötigen. Da die untersuchten Veränderungen, Kommutation und Transponierung, wieder eine dieser Partitionierungen ergaben, gelten auch hier die gleichen Kosten. Es konnte jedoch der im Vergleich niedrigere Kommunikationsaufwand bei der Blockpartitionierung belegt werden, welcher bei Transponierung der ersten Matrix noch

geringer ist. Diese Partitionierung ist daher für die Matrixmultiplikation unter dem Aspekt der Kommunikationskosten günstig.

Anschließend wurden von den betrachteten Partitionierungsarten die Zeilen- und Blockpartitionierung auf dem Postgres-XL-Cluster umgesetzt und eine Analyse von Anfrageplänen sowie Laufzeiten durchgeführt. Die Blockpartitionierung wurde dabei auf zwei Arten realisiert: zum einen mit der Vererbung zwischen Eltern- und Kindrelationen und zum anderen über den vorimplementierten `DISTRIBUTE BY`-Befehl. Dieser ermöglicht jedoch nur die Distribution über Hashing der Attributwerte, bei nicht anpassbarer Hashfunktion, oder die Round-Robin-Mechanik. Zur Lösung dieses Problems wurde eine Methode entwickelt, mit der Blockpartitionierung über den `DISTRIBUTE BY ROUNDROBIN`-Befehl realisierbar ist. Die Laufzeitbetrachtung führte zu der Erkenntnis, dass die `DISTRIBUTE BY`-Varianten wesentlicher weniger Zeit in Anspruch nahmen als die Vererbungsmethode. Bei den Matrixdimensionen $n = 900$ bis $n = 1800$ sind die Laufzeiten der Modulo-Variante und Blockpartitionierung über Round-Robin ähnlich. Ab der Matrixdimension $n = 3000$ verzeichnet jedoch die im Konzeptteil favorisierte Blockpartitionierung die beste Performance.

Auf Basis der gewonnenen Erkenntnisse lässt sich feststellen, dass Intraoperatorparallelität in Postgres-XL prinzipiell ohne Anpassung der sequentiellen SQL-Anfragen möglich ist. Jedoch zeigt sich auch, dass die Parallelisierung eines Operators stark von den Bestandteilen der SQL-Anfrage abhängt, da die Optimierung hier vielmehr auf der Ebene von einzelnen SQL-Operationen, als auf der gesamten mathematischen Operation, erfolgt.
Die durchgeführte Betrachtung der Kommunikationskosten für die verschiedenen Partitionierungsstrategien hat mit der Blockpartitionierung eine kostengünstige Strategie evaluiert, wodurch eine systemunabhängige Grundlage für weitere Arbeiten geschaffen wurde. Die Laufzeitanalyse bezog sich nur auf den Vergleich der einzelnen Partitionierungen und gibt eine Tendenz der unterschiedlichen Strategien. Durch die Berechnungszeiten des sequentiellen Plans bei der Vererbungsvariante wurde das Potenzial von den intraoperatorparallelisierten Anfragen der Modulo- und Round-Robin-Variante deutlich, was eine weitere Forschung in diese Richtung nahelegt.

Im Rahmen dieser Arbeit sind Fragestellungen offen geblieben oder entstanden, die Ansätze und Schwerpunkte kommender Arbeiten sein können. In der bisherigen Betrachtung wurden die SQL-Anfragen unverändert ausgeführt und damit die Erstellung des Anfrageplans gänzlich dem Datenbankoptimierer überlassen. Hier kann auf verschiedenen Wegen angesetzt werden. Zum einen ist das Anpassen der Anfrage möglich, um den Optimierer in die gewünschte Richtung zu lenken. Zum anderen wäre es realisierbar, die Anfrage aufzuteilen und von mehreren Nutzern parallel ausführen zu lassen. Im Fall der Matrixmultiplikation würde ein Nutzer einen Zielblock

berechnen. Als weiterer Ansatz kann die Betrachtung anderer paralleler Datenbanksysteme mit dem Ziel sein, dass der implementierte Datenbankoptimierer die Potenziale vielseitiger ausnutzt. Darüber hinaus wurden die Anfragen in dieser Arbeit auf - für Big-Data-Verhältnisse - relativ kleinen Daten und einem Cluster mit hoher Netzgeschwindigkeit durchgeführt. Es ist also zum einen zu prüfen, ob und wie sich die Erhöhung der Datenmenge auf die Anfragepläne sowie Laufzeiten auswirkt und im kritischen Bereich der Hauptspeichergröße agiert. In dieser Arbeit stand die Betrachtung der Anfragepläne und darauf basierenden Feststellung der grundsätzlichen Machbarkeit von Intraoperatorparallelität im Fokus. Zum anderen ist die Auswirkung langsamerer Netzgeschwindigkeiten, wie sie beispielsweise in verteilten Systemen vorzufinden sind, auf Anfragepläne und Laufzeiten zu beobachten.

Mit dieser Arbeit konnten erste positive Erkenntnisse im Bereich der Intraoperatorparallelität von Basisoperationen des wissenschaftlichen Rechnens mit SQL gewonnen werden, welche als Basis für kommende Forschung und Arbeiten auf diesem Gebiet dienen können.

7 Literaturverzeichnis

[2nd] 2ndQuadrant: *Postgres-XL*. `https://www.2ndquadrant.com/en/resources/postgres-xl/`. – abgerufen am 07.08.2017, auf CD

[BG83] Bernstein, Philip A.; Goodman, Nathan: Multiversion Concurrency Control - Theory and Algorithms. In: *ACM Trans. Database Syst.* 8 (1983), Nr. 4, 465–483. – DOI 10.1145/319996.319998. – auf CD

[Deo] Deolasee, Pavan: *How will Postgres-XL exploit the Parallel Query Capabilities of PostgreSQL 9.6?* `https://blog.2ndquadrant.com/how-will-postgres-xl-exploit-the-parallel-query-capabilities-of-postgresql-9-6/`. – abgerufen am 31.07.2017, auf CD

[DG04] Dean, Jeffrey; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004, 137–150. – auf CD

[Fis14] Fischer, Gerd: *Lineare Algebra*. Bd. 18. Springer Spektrum, 2014

[GBL+96] Gray, Jim; Bosworth, Adam; Layman, Andrew et al.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In: *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, 1996, 152–159. – auf CD

[GW97] Geijn, Robert A. d.; Watts, Jerrell: SUMMA: Scalable Universal Matrix Multiplication Algorithm. In: *Concurrency - Practice and Experience* 9 (1997), Nr. 4, S. 255–274. – auf CD

[HM00] Hameurlain, Abdelkader; Morvan, Franck: Invited Address: An Overview of Parallel Query Optimization in Relational Systems. In: *11th International Workshop on Database and Expert Systems Applications (DEXA'00), 6-8 September 2000, Greenwich, London, UK*, 2000, 629–634. – auf CD

[LM03] Langville, Amy N.; Meyer, Carl D.: Survey: Deeper Inside PageRank. In: *Internet Mathematics* 1 (2003), Nr. 3, 335–380. – DOI 10.1080/15427951.2004.10129091. – auf CD

[MH17] Marten, Dennis; Heuer, Andreas: Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements. In: *Open Journal of Databases (OJDB)* 4 (2017), Nr. 1, 22–42. https://www.ronpub.com/ojdb/OJDB_2017v4i1n02_Marten.html. – ISSN 2199–3459. – auf CD

[ÖV11] Özsu, M. T.; Valduriez, Patrick: *Principles of Distributed Database Systems, Third Edition*. Springer, 2011. – ISBN 978–1–4419–8833–1

[Pos] The PostgreSQL Global Development Group (Hrsg.): *PostgreSQL 9.6.3 Documentation*. The PostgreSQL Global Development Group. – auf CD

[PX] Postgres-XL: *Postgres-XL Overview*. www.postgres-xl.org/overview/. www.postgres-xl.org/overview/. – abgerufen am 07.08.2017, auf CD

[R C16] R Core Team; R Foundation for Statistical Computing (Hrsg.): *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2016. https://www.R-project.org/. – auf CD

[RJ86] Rabiner, L.; Juang, B.: An Introduction to Hidden Markov Models. In: *IEEE ASSP Magazine* 3 (1986), Jan, Nr. 1, S. 4–16. – DOI 10.1109/MASSP.1986.1165342. – ISSN 0740–7467. – auf CD

[SN95] Shatdal, Ambuj; Naughton, Jeffrey F.: Adaptive Parallel Aggregation Algorithms. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, 1995, 104–114. – auf CD

[Sql] *Information Technology - Database Language SQL.* : *Information Technology - Database Language SQL*, http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. – auf CD

[SSH13] Saake, Gunter; Sattler, Kai-Uwe; Heuer, Andreas: *Datenbanken - Konzepte und Sprachen, 5. Auflage*. MITP, 2013 http://www.it-fachportal.de/shop/buch/Datenbanken%20%E2%80%93%20Konzepte%20und%20Sprachen/detail.html,b174129. – ISBN 978–3–8266–9453–0

[Str03] Strang, Gilbert: *Lineare Algebra*. Springer-Verlag Berlin Heidelberg, 2003

[Thea] The Postgres-XL Global Development Group: *Postgres-XL 9.5r1.5 Documentation*, https://www.postgres-xl.org/documentation/index.html. – abgerufen am 12.06.2017, auf CD

[Theb] The Postgres-XL Global Development Group: *Postgres-XL 9.5r1.5 Documentation - What is Postgres-XL?* https://www.postgres-xl.org/

documentation/intro-whatis-postgres-xl.html. – abgerufen am 03.05.017, auf CD

[Weu16] Weu, Dennis: *Bachelorarbeit: Erkennung, Übersetzung und parallele Auswertung von Operatoren und Funktionen aus R in SQL*. 2016. – auf CD

Abbildungsverzeichnis

Programmbeispielverzeichnis

Tabellenverzeichnis

A Programmbeispiele

In den nachfolgenden Programmbeispielen wurden die Kostenschätzungen aus dem Anfrageplan entfernt, wobei die tatsächlichen Kosten weiterhin aufgeführt sind.

```
HashAggregate (actual time=858640.120..859035.012 rows=810000 loops=1)
Output: a_1."row", b_1.col, sum((a_1.val * b_1.val))
Group Key: a_1."row", b_1.col
->  Merge Join  (actual time=4668.431..335439.037 rows=729000000 loops=1)
    Output: a_1."row", a_1.val, b_1.col, b_1.val
    Merge Cond: (b_1."row" = a_1.col)
    ->  Sort (actual time=2280.701..2565.806 rows=810000 loops=1)
        Output: b_1.col, b_1.val, b_1."row"
        Sort Key: b_1."row"
        Sort Method: quicksort  Memory: 87858kB
        ->  Append (actual time=2.927..1536.674 rows=810000 loops=1)
            ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn
                _9) (actual time=1.528..1.528 rows=0 loops=1)
                Output: b_1.col, b_1.val, b_1."row"
            ->  Remote Subquery Scan on all (dn_1) (actual time=1.397..151.343 rows
                =90000 loops=1)
                Output: b_2.col, b_2.val, b_2."row"
            ->  Remote Subquery Scan on all (dn_2) (actual time=1.453..140.113 rows
                =90000 loops=1)
                Output: b_3.col, b_3.val, b_3."row"
            ->  Remote Subquery Scan on all (dn_3) (actual time=1.604..138.512 rows
                =90000 loops=1)
                Output: b_4.col, b_4.val, b_4."row"
            ->  Remote Subquery Scan on all (dn_4) (actual time=1.598..150.699 rows
                =90000 loops=1)
                Output: b_5.col, b_5.val, b_5."row"
            ->  Remote Subquery Scan on all (dn_5) (actual time=1.640..157.846 rows
                =90000 loops=1)
                Output: b_6.col, b_6.val, b_6."row"
            ->  Remote Subquery Scan on all (dn_6) (actual time=1.576..155.285 rows
                =90000 loops=1)
                Output: b_7.col, b_7.val, b_7."row"
            ->  Remote Subquery Scan on all (dn_7) (actual time=1.621..152.508 rows
                =90000 loops=1)
                Output: b_8.col, b_8.val, b_8."row"
            ->  Remote Subquery Scan on all (dn_8) (actual time=1.681..154.611 rows
                =90000 loops=1)
                Output: b_9.col, b_9.val, b_9."row"
            ->  Remote Subquery Scan on all (dn_9) (actual time=1.407..146.560 rows
                =90000 loops=1)
                Output: b_10.col, b_10.val, b_10."row"
   ->  Sort (actual time=2387.678..105477.829 rows=728999101 loops=1)
```

```
33        Output: a_1."row", a_1.val, a_1.col
34        Sort Key: a_1.col
35        Sort Method: quicksort  Memory: 87858kB
36        ->  Append (actual time=3.083..1594.636 rows=810000 loops=1)
37            ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn_
                  9) (actual time=1.691..1.691 rows=0 loops=1)
38                Output: a_1."row", a_1.val, a_1.col
39            ->  Remote Subquery Scan on all (dn_1) (actual time=1.390..157.847 rows
                  =90000 loops=1)
40                Output: a_2."row", a_2.val, a_2.col
41            ->  Remote Subquery Scan on all (dn_2) (actual time=1.419..136.356 rows
                  =90000 loops=1)
42                Output: a_3."row", a_3.val, a_3.col
43            ->  Remote Subquery Scan on all (dn_3) (actual time=1.129..137.392 rows
                  =90000 loops=1)
44                Output: a_4."row", a_4.val, a_4.col
45            ->  Remote Subquery Scan on all (dn_4) (actual time=1.698..163.664 rows
                  =90000 loops=1)
46                Output: a_5."row", a_5.val, a_5.col
47            ->  Remote Subquery Scan on all (dn_6) (actual time=1.661..171.967 rows
                  =90000 loops=1)
48                Output: a_6."row", a_6.val, a_6.col
49            ->  Remote Subquery Scan on all (dn_7) (actual time=1.636..162.999 rows
                  =90000 loops=1)
50                Output: a_7."row", a_7.val, a_7.col
51            ->  Remote Subquery Scan on all (dn_8) (actual time=1.463..146.201 rows
                  =90000 loops=1)
52                Output: a_8."row", a_8.val, a_8.col
53            ->  Remote Subquery Scan on all (dn_9) (actual time=1.367..155.463 rows
                  =90000 loops=1)
54                Output: a_9."row", a_9.val, a_9.col
55            ->  Remote Subquery Scan on all (dn_5) (actual time=1.586..162.319 rows
                  =90000 loops=1)
56                Output: a_10."row", a_10.val, a_10.col
```

Programmbeispiel A.1: Anfrageplan Matrixmultiplikation mit Vererbungsstrategie

```
Aggregate (actual time=2980.761..2980.761 rows=1 loops=1)
 Output: sum((alpha_a.val * val))
 -> Hash Join  (actual time=2978.714..2980.527 rows=900 loops=1)
     Output: alpha_a.val, val
     Hash Cond: ("row" = alpha_a."row")
     ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_6,dn_7,dn_8,dn_9)   (
         actual time=15.275..15.616 rows=900 loops=1)
         Output: val, "row"
     ->  Hash  (actual time=2963.138..2963.138 rows=900 loops=1)
         Output: alpha_a.val, alpha_a."row"
         Buckets: 1024  Batches: 1  Memory Usage: 47kB
         ->  Subquery Scan on alpha_a (actual time=2962.105..2962.859 rows=900 loops=1)
             Output: alpha_a.val, alpha_a."row"
             ->  HashAggregate  (actual time=2962.102..2962.473 rows=900 loops=1)
                 Output: col, sum(((alpha_a_1.val * val) * val))
                 Group Key: col
                 ->  Hash Join (actual time=2139.384..2450.474 rows=810000 loops=1)
                     Output: alpha_a_1.val, val, col, val
                     Hash Cond: (alpha_a_1."row" = "row")
                     ->  Hash Join (actual time=594.431..599.512 rows=900 loops=1)
                         Output: alpha_a_1.val, alpha_a_1."row", val, "row"
                         Hash Cond: ("row" = alpha_a_1."row")"
                         ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_
                             6,dn_7,dn_8,dn_9) (actual time=9.502..10.765 rows=900
                             loops=1)
                             Output: val, "row"
                         ->  Hash (actual time=584.877..584.877 rows=900 loops=1)
                             Output: alpha_a_1.val, alpha_a_1."row"
                             Buckets: 1024  Batches: 1  Memory Usage: 47kB
                             ->  Subquery Scan on alpha_a_1 (actual time
                                 =584.046..584.620 rows=900 loops=1)
                                 Output: alpha_a_1.val, alpha_a_1."row"
                                 ->  HashAggregate (actual time=584.042..584.298 rows
                                     =900 loops=1)
                                     Output: col, sum((sum(((val * val) * val))))
                                     Group Key: col
                                     ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn
                                         _4,dn_5,dn_6,dn_7,dn_8,dn_9) (actual time
                                         =565.134..571.146 rows=8100 loops=1)
                                         Output: col, sum(((val * val) * val))
                     ->  Hash (actual time=1537.099..1537.099 rows=810000 loops=1)
                         Output: col, val, "row"
                         Buckets: 1048576 (originally 524288)  Batches: 1 (originally
                             1)  Memory Usage: 49325kB
                         ->  Remote Subquery Scan on all (dn_1,dn_2,dn_3,dn_4,dn_5,dn_
                             6,dn_7,dn_8,dn_9)  (actual time=5.093..485.853 rows=810000
                              loops=1)
                             Output: col, val, "row""
```

Programmbeispiel A.2: Anfrageplan Vorwärtsalgorithmus

B Tabellen

Tabelle B.1 zeigt die gemessenen Laufzeiten (in ms) der in 5.2 beschriebenen Matrixmultiplikationsanfrage auf den in 5.4.1 erläuterten Partitionierungsstrategien. Bei der Vererbungsvariante wurde die Anfrage der Dimension $n = 1500$ nach 7 Stunden ohne Ergebnis abgebrochen.

Dimension	Testlauf	Blockweise, via Round-Robin	Zeilenweise, via Modulo	Blockweise, via Vererbung
900	1	101430	75732	864343
	2	92559	74079	845751
	3	87210	73706	859167
	4	89599	78778	854703
	5	90844	80858	836787
1200	1	250864	196452	2298860
	2	211175	187657	2274921
	3	220548	184114	2391932
	4	223489	206367	2293283
	5	229691	184287	2289321
1500	1	466972	353116	$>$ 7 Stunden
	2	448788	362778	-
	3	448619	395833	-
	4	439357	381367	-
	5	430899	367907	-
1800	1	762158	627659	-
	2	794796	604945	-
	3	751398	641511	-
	4	766801	654172	-
	5	763338	639481	-
3000	1	2125039	2840625	-
	2	2138117	2705397	-
	3	2119178	2890165	-
	4	2148988	2859685	-
	5	2159162	2749166	-

Tabelle B.1: Laufzeiten (in ms) der Matrixmultiplikation bei verschiedenen Partitionierungen

Die nachfolgende Tabelle B.2 stellt die Laufzeiten (in ms) des Vorwärtsalgorithmus auf einer Blockpartitionierung (Round-Robin Variante) für die Sequenzlängen 5 bis 10 dar.

Dimension	Testlauf	5	6	7	8	9	10
900	1	6999	8619	10300	11137	14360	15907
	2	6655	8098	9388	13444	13945	16425
	3	6899	8457	10396	12904	12638	15882
	4	6589	8560	11096	12506	13988	15745
1200	1	11390	15083	17523	20341	21869	27030
	2	10690	13335	17354	19928	22749	25537
	3	9939	13241	15808	19460	22041	25093
	4	9786	12590	16344	19849	21781	26792
1500	1	17755	20731	26791	30321	33028	41100
	2	14925	20208	25692	29831	34852	39056
	3	16284	19912	24535	29850	34543	41355
	4	15481	20160	23479	29650	33439	37728
1800	1	21745	27982	39233	41889	48546	54863
	2	20697	30318	34746	43316	50897	55884
	3	20652	29867	33564	41097	47320	58064
	4	20892	28816	36173	42371	49286	56821
3000	1	57851	76977	96774	112691	133931	145814
	2	59242	76842	97905	111234	133771	149935
	3	58480	77098	96055	113254	131909	153281
	4	57236	76252	96832	113281	132394	147931

Tabelle B.2: Laufzeiten (in ms) des Vorwärtsalgorithmus

C Datenträgerübersicht

Dieser Masterarbeit liegt eine CD bei, deren Struktur hier aufgelistet ist.

- Text (Ordner)
  - beinhaltet die PDF-Datei der Masterarbeit
- Literatur (Ordner)
  - beinhaltet die im Literaturverzeichnis gekennzeichnete Literatur
  - Dateien sind nach dem Schema [Zitierschlüssel]-titel-der-literatur.pdf benannt
- Code (Ordner)
  - Testscripte (Ordner)
    - beinhaltet zum Testen verwendete .sh-Dateien
  - Testergebnisse (Ordner)
    - beinhaltet die Testergebnisse als .csv-Dateien und die Plots als .pdf-Dateien
  - Erstellungsscripte (Ordner)
    - beinhaltet zum Generieren der Testmatrizen verwendete .py- und .r-Skripte
  - Validierung (Ordner)
    - beinhaltet den zur Validierung genutzten Vorwärtsalgorithmus als .r-Programm
    - beinhaltet das zur Validierung der Matrixmultiplikation genutzte .r-Programm

Selbstständigkeitserklärung

Hiermit erkläre ich, Dennis Weu, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen, direkt oder indirekt, übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift